NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

OBJECT-ORIENTED PROGRAMMING: AN ASSESSMENT OF FUNDAMENTAL CONCEPTS AND DESIGN CONSIDERATIONS

by

Alan Lee Fink

March 1992

Thesis Advisor: Michael L. Nelson

Approved for public release; distribution is unlimited.

CURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
a SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution is unlimited
b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
. PERFORMING ORGANIZATION REPORT NUMBER(S) Monterey, Ca 93943-5000	5. MONITORING ORGANIZATION REPORT NUMBER(S)
a. NAME OF PERFORMING ORGANIZATION Computer Science Dept. Naval Postgraduate School; 6b. OFFICE SYMBOL (if applicable) CS	7a. NAME OF MONITORING ORGANIZATION Naval Postgraduate School
c. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000	7b. ADDRESS (City, State, and ZIP Code) Monterey, CA 93943-5000
a. NAME OF FUNDING/SPONSORING ORGANIZATION; 8b. OFFICE SYMBOL (if applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
c. ADDRESS (City, State, and ZIP Code)	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO

1. TITLE (Include Security Classification)
OBJECT-ORIENTED PROGRAMMING: AN ASSESSMENT OF FUNDAMENTAL CONCEPTS AND DESIGN CONSIDER-ATIONS (U)

2. PERSONAL AUTHOR(S)
ink, Alan Lee

3a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Master's Thesis	FROM _ TO _	March 1992	162

. SUPPLEMENTARY NOTATION The views expressed in this thesis are those of the author and do not reflect the fficial policy or position of the Department of Defense or the United States Government.

COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Object-Oriented Analysis Object-Oriented Design
			Object-Oriented Programming Object-Oriented Programming Language

. ABSTRACT (Continue on reverse if necessary and identify by block number)

The latest buzzword penetrating the professional computer science literature is Object-Oriented Programming. omputer scientists extol its theoretical virtues while developers explore its potential for streamlining the process of ftware development. Amidst all this activity there remains substantial confusion about fundamental concepts and e programming language mechanisms which implement these concepts. Too often, students of object-oriented pro-ramming mistake proficiency in an object-oriented language for efficient application of object-oriented techniques. he immediate consequence is poorly conceived, sometimes conflicting, efforts at exploiting reusability, information ding and other object-oriented capabilities.

This thesis reviews the benefits attributed to object-oriented programming, arrives at definitions for fundamental ncepts, advances recommendations for conducting object-oriented analysis and object-oriented design, and reviews me tradeoffs which designers need to consider when developing object-oriented classes and hierarchies.

DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
a. NAME OF RESPONSIBLE INDIVIDUAL MAJ Michael L. Nelson	22b. TELEPHONE (Include Area Code) (408) 646-2026; 22c. OFFICE SYMBOL CS/NE

D FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete — SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED

OBJECT-ORIENTED PROGRAMMING: AN ASSESSMENT OF FUNDAMENTAL CONCEPTS AND DESIGN CONSIDERATIONS

by

Alan Lee Fink
Lieutenant, United States Navy
B.A., Philosophy, Maryland University, 1979
B.A., Economics, Maryland University, 1979
M.S., Mineral Economics, Pennsylvania State University, 1983

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1992

ABSTRACT

The latest buzzword penetrating the professional computer science literature is *Object-Oriented Programming*. Computer scientists extol its theoretical virtues while developers explore its potential for streamlining the process of software development. Amidst all this activity there remains substantial confusion about fundamental concepts and the programming language mechanisms which implement these concepts. Too often, students of object-oriented programming mistake proficiency in an object-oriented language for efficient application of object-oriented techniques. The immediate consequence is poorly conceived, sometimes conflicting, efforts at exploiting reusability, information hiding and other object-oriented capabilities.

This thesis reviews the benefits attributed to object-oriented programming, arrives at definitions for fundamental concepts, advances recommendations for conducting object-oriented analysis and object-oriented design, and reviews some tradeoffs which designers need to consider when developing object-oriented classes and hierarchies.

F75
c.1

TABLE OF CONTENTS

LIST OF FIGURES

TABLE OF ABBREVIATIONS

ADT	Abstract Data Type
CAD	Computer Aided Design
CLOS	Common Lisp Object System
DFD	Data Flow Diagram
ER	Entity Relationship
MI	Multiple Inheritance
OO	Object-Oriented
OOA	Object-Oriented Analysis
OOD	Object-Oriented Design
OOP	Object-Oriented Programming
OOPL	Object-Oriented Programming Language

ACKNOWLEDGEMENTS

The writer wishes to recognize the efforts made by his adviser, MAJ Michael L. Nelson, to protect society from the dissemination of false doctrine and weak interpretation. Dr. Robert B. McGhee, the second reader, impressed upon the writer the importance of testing theoretical conclusions with actual implementations. Finally, the writer recognizes with gratitude the debt owed to his recently deceased father, Donald A. Fink. It was from his father that the writer acquired certainty of purpose, intellectual curiousity, and analytic discipline.

I. INTRODUCTION

Acolytes of object-oriented (OO) concepts and methodologies maintain that this new approach to software development promises to be more than just another set of programming languages. Rather, it represents a comprehensive philosophy for planning, designing, and implementing solutions to complex problems. An explosion of articles, magazines, books, and conferences dedicated to exploring and popularizing OO techniques occurred during the 1980's. This concentration of effort notwithstanding, the OO philosophy has yet to arrive at a consolidated point-of-view on many fundamental concepts.

The objective of this thesis is to review the potential benefits of OO techniques, to define fundamental OO concepts, to advance recommendations for organizing OO design, and to demonstrate conflicts among OO mechanisms that dilute the favorable properties ascribed to object-oriented programming (OOP).

The beneficial properties of the OO approach to software development are briefly reviewed in this chapter. These properties are commonly viewed as desirable from the perspective of software engineering. Not all the properties are unique to the OO philosophy, much less common to all object-oriented programming languages (OOPLs). Nevertheless, the OO philosophy represents a comprehensive attempt at integrating all the concepts and facilities affecting the software lifecycle in a fashion that enhances the desirable properties.

A. COMPLEXITY MANAGEMENT

Software development increasingly occurs in an industrial setting typified by product complexity, system longevity, and incessant product evolution. (Jacobson, 1991) OO techniques have been employed for developing complex software products such as compilers, databases, computer aided design (CAD) systems, simulations, meta models, operating systems, spreadsheets, signal processors, and control systems. (Rumbaugh, 1991) Development of such complex systems requires architectures, methods, and processes that divide system development into smaller parts and that can handle change efficiently. (Jacobson, 1991) The following subsections list desirable software features which the OO approach to software development attempts to innovate, thus making it well suited for managing complex application problems.

1. Abstraction

> *Abstraction is used to simplify the design of a complex system by reducing the number of details that must be considered at the same time.* (Berzins, 1991, pg. 79)

Abstraction is an intellectual process that facilitates the comprehension of complex entities and processes through simplification. This simplification allows knowledge to be expressed as generalized, essential information which can be absorbed and understood by human beings.

The level of detail necessary to formulate an abstraction varies with the requirements of the problem. (Booch, 1987) In the context of OO analysis, design, and programming, abstraction focuses attention on the behaviors and attributes of objects rather than on the implementation details (which vary from one language to another). This method of thinking about problem entities allows problems to be pursued as successive refinements, with each refinement constituting an abstraction manifesting a particular level of detail. Hence, designs and programs can be conceived of as multileveled structures of abstractions.

2. Information Hiding

> *Information hiding emphasizes the need to separate function from implementation. Apart from continuity, it is also related to the requirements of decomposability, composability and understandability: to separately develop the modules of a system, to combine various existing modules, or to understand individual modules, it is indispensable to know what each of them may and may not expect from the others.* (Meyer, 1988, pg. 23)

Information hiding[1] is a technique for minimizing interdependencies among separately written modules. (Snyder,

[1]Sometimes termed encapsulation in the OO literature.

1986) Knowledge about data structure and function implementation is kept private by forcing interaction with these structures through an external interface. In the wider context of software development, information hiding promotes the independent construction of cooperating modules, and, most importantly, isolates the effects of implementation modification to the affected module. So long as external interfaces are stable, implementation details can be modified without impinging upon users of the interface. Hence, software maintenance becomes localized, no longer a perilous search for linkages in interrelated program modules, with consequent economies in time, human, and onetary resources.

Information hiding is a critical metric of any OOPL. Consequently, individual OOPLs should be studied to determine the degree to which information hiding is enforced.

3. **Reusability**

> *Reuse may be defined as the effective ability to incorporate objects created for one software system into a different software system. The essence of reuse is the ability to take all or part of a product and completely and correctly embed it within a new product that may be constituted and structured quite differently.* (Wasserman, 1991, pg. 55)

Reusability clearly coincides with 'why reinvent the wheel' modes of thinking. Even limited software development experience is enough for one to notice that programs exhibit pervasive commonalities with respect to algorithms, data structures, and functions. Reusability is a language property that allows previously developed software to be readily incorporated into new software. Two substantial benefits flow from reuse: (1) development effort is reduced; and, (2) reused code has (presumably) been tested and verified[2].

The principal OO mechanisms for achieving reuse are inheritance, polymorphism, and dynamic binding. Much of the value of programming in the OO environment arises from the capability to use previously developed code stored in software libraries. Developers may also be familiar with a problem's requirements and important abstractions; consequently, opportunities for reusing not only software, but entire designs and requirements also exist. (Booch, 1991)

4. Extendibility

Extendibility is the ease with which software products may be adapted to changes in requirements. (Meyer, 1988, pg. 5)

Extendibility is a concept allied to, but distinct from, reusability. It encompasses those properties/mechanisms which enable new code to be developed as extensions to previously written code. Extendibility assumes greater importance as problem understanding improves, resulting in new requirements. As program scale grows, extendibility is best achieved through design simplicity and modular

[2]This does not, however, relieve developers of the burden of ensuring that borrowed code still functions properly in its new environment. (Perry, 1990)

decentralization. (Meyer, 1988) In the OO environment, extendibility is realized through the application of inheritance techniques to class definitions in class hierarchies.

5. Maintainability

> *A designer endeavors to organize a design so that it is resilient to change; a packaging that will remain stable over time is sought. The answer is to separate those parts of the system that are intrinsically volatile from those parts that are likely to be stable.* (Coad and Yourdon, 1991, pg. 15)

Maintainability is principally an economic concept. It refers to the efficiency by which modifications can be introduced over the software lifecycle. Technically, maintainability concerns the degree to which linkages in program elements magnify the effects of modifications. Economically, maintainability reflects the cost required to correct bugs, to modify code, or to extend code. The primary cost is the human effort required to police change. Software which collectively exhibits strong abstraction, information hiding, reusability, and extendibility generally manifest favorable maintainability properties.

B. THESIS MOTIVATION

In point of fact, the lack of a standardized conceptual foundation for many OO ideas has resulted in an uneven record with respect to the completeness with which individual OOPLs contribute toward achieving a unified realization of the beneficial software properties. One of the purposes of this thesis is to indicate the mapping of fundamental OO concepts to abstraction, information hiding, reusability, extendibility, and maintainability. Given these associations, it is possible to investigate how the interactions of language mechanisms which engineer the fundamental OO concepts can mitigate the desirable properties should potential conflicts not be thoroughly understood. A primary objective, therefore, is to facilitate better OO software development by acknowledging that conflicts exist which must be accounted for during analysis, design, implementation, and maintenance.

C. THESIS ORGANIZATION

Chapter II surveys the OO literature in attempting to formulate definitions for fundamental OO concepts. Chapter III draws upon the OO literature to advance recommendations for conducting object-oriented analysis (OOA) and object-oriented design (OOD). Chapter IV examines connections between functional decomposition and subclass responsibility. Chapter V furthers the aims of Chapter III by investigating two major OO problems: (1) conflicts between information hiding and inheritance mechanisms; and, (2) design conflicts between composition and inheritance. Finally, Chapter VI offers conclusions and suggestions for further research.

II. FUNDAMENTAL OBJECT-ORIENTED CONCEPTS

OO methodology is occasionally presented as a Kuhnian paradigm by which is meant a corpus of scientific work which "...defines the legitimate problems and methods of a research field for succeeding generations of practitioners." (Kuhn, 1962, pg. 10) Budd accepts this departure point, adding that the OO paradigm "...forces us to reconsider our thinking about computation, about what it means to perform computation, about how information should be structured within a computer." (Budd, 1991, pg. 3) Essentially, then, the OO paradigm is about knowledge representation in the computer environment. From this perspective, OO methodology constitutes a new way of conceptualizing and solving the problems of software engineering.

Although there currently exist no OO standards for OO language designers to observe, considerable consensus exists as to the primary concepts which formulate the OO paradigm. Nevertheless, there remains sufficient semantic variation in the literature to warrant a thorough review of these concepts. The balance of this chapter surveys the OO literature with the intent of arriving at definitions of fundamental concepts for use in succeeding chapters.

A. OBJECTS

Objects occupy a curious dual status. Often, they are introduced in anthropomorphic terms as the entities (physical or ideational) in the problem domain. Alternatively, they are presented as the primary programming constructs in OO languages; constructs which 'closely' parallel the entities in the problem domain. Although the differences between the two views are minimal in terms of practical consequences for OO analysis and design, the distinction should be bourn in mind since the latter emphasizes that objects are constrained not only by their real-world possibilities but also by the capabilities of computers and programming languages.

1. Definition

> *Objects encapsulate both state and behavior.* (Halbert and O'Brien, 1987, pg. 72)

> *Objects are entities that combine the properties of procedures and data since they perform computations and save local state.* (Stefik and Bobrow, 1986,pg. 41)

> *An object has state, behavior, and identity; the structure and behavior of similar objects are defined by their common characteristics; the terms instance and object are interchangeable.* (Booch,1991, pg. 77)

> *Objects are autonomous entities that respond to messages or operations and share a state.* (Wegner,1987, pg. 168)

Several salient points can be drawn from these definitions. First, objects have a structure (representation), which preserves the state of an object. An object may manifest many states over the course of its existence (i.e., its state may change); hence, objects can have a history. Second, objects exhibit observable behavior. Objects communicate with one another by passing messages to elicit needed behavior[1]. Third, objects have, in some sense, an existential status that uniquely distinguishes each object from all others. This status is usually termed identity. Fourth, the fact that an object has an identity and can have a history implies that it can exist beyond the lifetime of the program(s) in which the object may have been created or used. This quality is termed persistence. (Loomis, 1991)

A few clarifications are in order. Objects can be created that do not have a structure However, it is difficult to isolate what distinguishes these entities from procedures, and calls into question the existential status of such entities - that is, must not there be a 'some' for an object to be something? Behavior is the set of actions an object can undertake. In the context of a program, an object sends a message to another object (or itself) requesting a

[1]Message-passing is one mechanism for managing object communication. Messages are sent "...to an object to tell it to perform one of its methods." (Nelson, 1991, pg. 4) Messages are essentially legal invocations of methods (behaviors) associated with objects. Legal invocations are those made to methods which objects have made available. Note that not every OOPL utilizes message-passing as the mechanism for inter-object communication. CLOS (Keene, 1989), for example, uses generic function calls. This sometimes leads to what is called the "message passing paradigm" in which "...the only way to access an object or any of its variables is by sending it a message." (Nelson, 1991, pg. 4)

service offered by the receiving object. The receiving object determines how best to comply with the request, selecting among a set of methods (operations) which satisfy the request, responding in a form that the sender can understand and use. Hence, there is a semantic quality to behavior. Note that much of the versatility and confusion in OO programming arises from the mechanisms that determine which object receives a request and which method is selected.

2. Identity

As stated in the introduction to this chapter, the OO paradigm constitutes a new approach to representing knowledge in the computer environment. Specifically, objects, as programming constructs, achieve a sufficiently elevated level of abstraction so as to closely parallel their real-world counterparts. Real-world entities are bounded and distinguishable. In the computer environment this requires "...the ability to distinguish objects from one another regardless of their content, location or addressability, and to be able to share objects. Object identity enables us to realize this goal." (Khoshafian and Copeland, 1986, pg. 406)

The practical consequence is that an OO language should preserve this existential status as an independent fact about objects. This existential status can be described as the space (which is relocatable) in any memory which the object happens to occupy. It can be argued that an OO language must maintain identity despite changes in an object's state, address, or user-defined name, and throughout its lifetime. (Khoshafian and Copeland, 1986) An OO language accomplishes this best by maintaining a built-in identifier for an object that (presumably) does not change. "The failure to recognize the difference between the name of an object and the object itself is the source of many kinds of errors in object-oriented programming." (Booch, 1991, pg. 84) These errors include assignment operations which orphan objects[2], aliasing through assignment (structural sharing), and inappropriate semantics for equality operators. (Booch, 1991)

3. **Persistence**

An object is created, persists for some period of time, and can be destroyed. Its lifetime may be less than, parallel to, or exceed that of the program in which it is created. This last possibility represents a database issue in general, but with particular refinements for objects; the identity and nature (class of which it is an instance) of an object must be preserved along with its state.

[2]For example, the assignment *object_x := object_y* renders inaccessible the original object for which *object_x* was the name.

4. Conceptual Distinctions

a. Objects and Programs

The OO paradigm leads the programmer to an entirely different perspective on program construction. What has been described as the traditional imperative approach to programming consists of procedural modules which act on data. This perspective leads to a top-down, functional decomposition of programs. OO programs consist of objects acting cooperatively, but autonomously. Cooperative behavior can achieve various levels of integration, producing system behavior at the highest levels. The great strength of this approach is the comparative ease with which complex systems can be modeled as interacting objects.

b. Objects and Data

Objects are not simply data structures. It is important to recognize that objects are entities which manifest both structure and behavior. Moreover, the implementation of an object's structure should be hidden from other objects (i.e., encapsulated) for the reasons outlined in Chapter I. This is not the case for data driven programs in which data structures are globally accessed and modified.

c. Objects and Equality

Object identity necessitates reviewing in what respect objects can be said to be equal. Database applications may require that relational comparisons be available. (Nelson, Moshell, and Orooji, 1990) Most OO languages allow programmers to generate operations for determining equality. Variations include reference identity in which references point to the same object, and structural identity in which corresponding parts of objects have the same value. It is worth noting that the semantics of equality and relational operators in this context do not necessarily follow the use of normal logic - objects can at once be not greater than, not equal to, and not less than one another[3]. (Nelson, Moshell, and Orooji, 1990)

B. CLASSES

A major milestone in the evolution of software engineering has been the modularization of software components. Berzins and Luqi define a module as a "...conceptual unit in a software system that corresponds to a clearly identifiable region of the program text." (Berzins and Luqi, 1991, pg. 13) From this perspective, modularization is a facet of software construction which produces "...software systems made of autonomous elements connected by a coherent, simple structure." (Meyer, 1988, pg. 11)

Actual modular elements will vary among programming languages; the essential aspect of modularization is that it

---

[3]Nelson, Moshell, and Orooji suggest that there are contexts in which "...it is inappropriate to say that one object is greater than or less than another." (Nelson, Moshell, and Orooji, 1990, pg. 321)

promotes conceptual localization of code. This localization can be exploited to realize other desirable software properties including data abstraction, encapsulation, reusability, extendibility, reliability, and maintainability.

Classes are the key modules in most OO languages. Note that this position is taken in view of the fact that objects are declared as instances of classes. It is perhaps more accurate to state that classes are the key design modules and objects, as instances of classes, are the key program modules.

Most OO languages provide for classes, although a few do not. The various languages utilizing classes do so for different purposes, depending upon their overall philosophy. This section surveys the differing uses for classes.

1. Definition

A class is a template (cookie cutter) from which objects may be created by 'create' or 'new' operations. Objects of the same class have common operations and therefore uniform behavior. (Wegner, 1987, pg. 169)

A class is a description of one or more similar objects. In comparison to procedural programming languages, classes correspond to types. (Stefik and Bobrow, 1986, pg. 42)

Whereas an object is a concrete entity that exists in time and space, a class represents only an abstraction, the 'essence' of an object, as it were. (Booch, 1991, pg. 93)

A class should allow you to build a taxonomy of objects on an abstract, conceptual level. (Wirfs-Brock, Wilkerson, and Wiener, 1990, pg. 22)

An object manifests structure and behavior. Groups of objects may display common structure and behaviors which can be abstracted into class definitions[4]. Once an object is identified (declared) as belonging to a particular class, its structure and behavior can be delimited according to that common definition. This contributes powerfully toward reusability: individual objects acquire definition from an existing class 'template'.

A class typically consists of variables whose values are either defined individually for each object (instance variables), or are defined in the class definition itself as belonging to all instances (class variab es), and of methods which define behavior appropriate to instances of the class. Some of the methods in a class are auxiliary in the sense that they carry out operations needed by other methods, but do not themselves correspond to abstractions which are observable.

[4]Odell presents a philosophical analysis of classes that begins with describing cognitive categories for discussing knowledge and knowledge acquisition. (Odell, 1991) His principle point is that humans formulate their understanding of objects as concepts, and it is these concepts which are used to build classes. Concepts can have a name (representative symbol), an intension (definition), and an extension (the group of objects the concept applies to). Note that this way of viewing classes does not require that OOP objects parallel real-world objects so much as that they parallel human cognition of these objects. Moreover, this view also lends itself to defining 'unreal' or imaginary objects.

These auxiliary methods are a form of knowledge about class instances which users do not need to know.

An interface to a class consists of those variables and methods which are visible to other objects and to subclasses (defined in Chapter II/Section 3)[5]. The interface available to other objects is called the "external view" and the interface available to subclasses is called the "internal view."[6] (Micallef, 1988, pg. 13)

Not every OOPL offers a means for limiting the various interface visibilities. Those that do (C++, for example) provide mechanisms for enforcing private and public distinctions. Private variables and methods represent knowledge that is not known by other objects, and therefore are not part of the external or the internal interface. Public variables and methods are known and integrated into the external and internal interfaces. Generally, encapsulation is best enforced if variables are kept private, accessible only through public methods. An additional level of control can be applied to the internal interface: variables or methods can be

---

[5]An interface can also be viewed as the set of behaviors an object makes available for use by other objects.

[6]Note that a third interface to new objects is sometimes mentioned in the OO literature. (Micallef, 1988) This interface involves decisions about the external interface, initialization procedures, and class variables which will be made available to instantiations of a class. Also, the various views are called interfaces (for example, the external view is the same as the external interface).

declared protected which renders them visible to subclasses but not to other objects[7].

As previously noted, classes also constitute the key modules in most OO languages. Class modularity serves the objectives stated above and will be explored in the following subsections. The capability of classes to serve as a pattern for object declaration ensures consistent realization of these objectives in an OO program.

2. Classes as Abstract Data Types

Coad and Yourdon quote the Dictionary of Computing (Oxford University Press, 1986) in defining data abstraction as "...the principle of defining a data type in terms of the operations that apply to objects of the type, with the constraint that the values of such objects can be modified and observed only by the use of the operations." (Coad and Yourdon, 1991, pg.7) Meyer notes that in describing data structures it is desirable to have complete, precise, unambiguous descriptions that are not based on the physical

---

[7]In C++, there is an interesting split in control over the internal interface. (Atkinson and Atkinson, 1991) A superclass may declare class features to be private, protected, or public. However, it is the subclass which determines how public, private, and protected visibility declarations are inherited. The visibility of inherited features can be declared public, private, or protected. The declaration mechanism essentially allows a subclass to redeclare inherited public features as protected or private, and inherited protected features as private. Inherited private features remain private to the superclass - subclasses as well as users do not enjoy access privileges to these features.

representation of the underlying structure. (Meyer, 1988) An abstract data type (ADT) specifies a class of data structures "...not by implementation, but by the list of services available on the data structures, and the formal properties of these services." (Meyer, 1988, pp. 53-54)

The data structure and associated services of an ADT conceptually form a unified whole. The separation of visible services from implementation details, central to the notion of an abstract data type, is accomplished by an interface which describes the services which the type performs. Consequently, a user of a data type understands the type as a closed description of behavior, the successful use of which requires only knowledge of the interface.

In many OO languages, such as Eiffel (Meyer, 1988) and C++, classes are equivalent to ADTs. Hence, classes, the modular units of interaction, assume a specific purpose: the description of data types. The interactions between modules (classes) are managed through the type interfaces. Nevertheless, it is important to understand that the principle function of classes is to serve as templates for object instantiation and not as predicate descriptors. (Wegner, 1988) This critical difference assumes significance when considering inheritance (described in the next section) in which classes maintain set/subset relationships to facilitate code sharing. The mechanisms for inheritance do not always meet the strict requirements for type/subtype behavioral compatibility which the class/type equivalence requires.

3. Classes and Encapsulation

Although classes specify the behavior of data types, as modular software components they also embody the implementation details of structure and behavior. Modularization permits the design of interfaces which encapsulate these implementation details, thereby achieving the many benefits attributed to information hiding in Chapter I. The class interface need not include auxiliary methods, further increasing encapsulation. Hence, the class interface not only describes services available to users, but limits the ability of users to directly know, access, or modify the actual data structure of a class.

Encapsulation represents a property, but not a responsibility of classes. Programmers must specifically design interfaces which segregate implementation from specification, and the OO philosophy of the language must support encapsulation by restricting access/manipulation of data structures to the designed interface. It is curious that not every OO language supporting classes enforces encapsulation as described. Instance variables in Simula, for example, are directly accessible. (Micallef, 1988) For reasons adduced in Chapter I, this undesirable condition increases the linkages among program modules, diluting the reliability of code, and increasing the difficulties attendant to maintenance.

4. Categories of Classes

Within the context of software system design, classes serve varying purposes depending upon the requirements of the problem and the nature of the knowledge being modeled. These class roles generally reflect design decisions about how knowledge of data types should be distributed in the evolving class structure.

a. Abstract Classes

Abstract classes appear at the higher levels in inheritance hierarchies (discussed in the next section). Jumping ahead, an abstract class serves as a placeholder of methods and data common among the descendant classes in a hierarchy. (Wu, 1991) Hence, an abstract class serves as a repository of knowledge held in common by all of its descendent classes. As such, abstract classes are not intended to serve as templates for object instantiation. Rather, they eliminate the duplication of knowledge among related types, thereby reducing coding complexity and facilitating testing, debugging, and maintenance. Additionally, abstract classes also promote the design of a common protocol (interface) among the related types. This figures importantly in languages emphasizing polymorphism.

b. *Concrete Classes*

Concrete classes are simply classes that are intended to serve as templates for object instantiation. Note that a concrete class may or may not be descended from an abstract class. However, it has been suggested that it is possible that abstract classes should not be descended from concrete classes. (de Paula and Nelson, 1991)

c. *Virtual Classes*

Virtual classes (C++ terminology) contain virtual methods. Virtual methods are those methods in a class whose implementations may be overridden (redefined) in descendent classes. This allows method names and signatures to be shared (see Section E on polymorphism). It should be noted, however, that many OO languages do not include this concept - any method of any class may be overridden in descendent classes.

d. *Pure Virtual/Deferred Classes*

Classes may exhibit behavioral commonalities that designers want to place into abstract classes without forcing a common implementation. Pure virtual classes (C++ terminology) allow for the specification of interfaces, part or all of whose methods are to be implemented in descendent classes[8].

---

[8]The concept of pure virtual functions should not be confused with the concept of subclass responsibility. Pure virtual functions are employed to ensure such methods form part of the interface of descendent classes. Subclass

Meyer refines the concept of pure virtual classes, arguing that such deferred classes (Eiffel terminology) additionally require the specification of logical pre/post conditions for deferred implementations. (Meyer, 1988) Although correct in principle, it is difficult to discern how this complication assists design since a behavior is elevated to virtual status precisely because it is shared by descendants, and should therefore demonstrate the same semantic properties in descendent implementations.

d. Parameterized Classes

Parameterized classes are a form of generic structures. Such classes provide methods which operate on data structures whose types are not completely defined. An example would be a tree class for which the types of individual nodes are undefined. Another example would be an array class containing elements whose types are undefined. Current OO languages do not offer parameterized classes, although C++ designers are currently developing the technique for eventual introduction. (Budd, 1991)

---

responsibility derives from the notion of functional decomposition (see Chapter IV). Specifically, subclass responsibility involves decisions about the distribution of methods among ancestor and descendent classes in cases where ancestor methods rely on the knowledge that descendants must implement certain methods.

e. *Metaclasses*

Abstract, concrete, and virtual classes represent roles which classes can assume in the context of program design. In this capacity, classes continue to serve as patterns for instantiation. Some OO languages (including Smalltalk, for example) carry through the object point-of-view to include all constructs in the language space. Consequently, classes must be treated as objects as well. This raises the requirement to create, initialize, and destroy classes. To accommodate these needs, such languages provide for metaclasses - classes from which class objects are instantiated. "A class object is typically the only instance of a metaclass." (Budd, 1991, pg. 376)

Metaclasses are a conceptual complexity which are, mildly expressed, difficult to understand. It is difficult to place an end to the recursion implicit in defining everything as an object. For example, consider this obscure passage from Budd describing metaclasses in Smalltalk:

> *The class Class is a subclass of the class Object; and thus, the object Class points to the object Object as its superclass. On the other hand, the object Object is an instance of the class Class; and thus, Object points back to class. Class Class is itself a class, and thus an instance of itself.* (Budd, 1991, pg. 265)

Classes force instances to exhibit the same behavior; thus, metaclasses force classes to exhibit the same behavior. Metaclasses allow for class instances which specialize the behavior of other classes. As an example, a class may need to override the constructor for an object defined by another class. The corresponding metaclass would allow for the overriding of the other class' constructor.

Carried to its logical extreme, the idea that all constructs in the programming environment are objects requires that "...the metaclass must be considered an object in its own right, and is therefore created by the metametaclass, which is in turn created by the metametametaclass, etc." (Nelson, 1990, pg. 7) Nelson points out that most OOPLs supporting metaclasses ignore this problem, or simply declare metaclasses to be special objects provided by the system. (Nelson, 1990)

Metaclasses clearly provide a higher level of abstraction; nevertheless, they also move away from the real-world parallelism that OO languages accentuate, forming a strange dual definition for classes, and are perhaps best relegated to theoretical discussions.

5. Epistemological Issues

a. Objects of Knowledge or Objects of Belief?

An interesting side issue is the epistemological status of classes. Articles considering class design ordinarily contain a seemingly harmless footnote to the effect that design teams should possess at least one subject matter expert who presumably fully understands the problem domain from both a theoretical and experiential perspective which supersedes the immediate application problem. At issue is whether the knowledge embodied by classes must be justified. As an example, an application modeling certain kinds of planetary phenomena may start from a Ptolomeic or a Copernican explanation of behavior and achieve reliable behavioral results. Yet, only the Copernican theory is justified as knowledge. If one objective of class design is to mimic the real-world, then designers should be required to justify to some degree the knowledge represented by classes. Too often in application design, belief is substituted for knowledge; much to the detriment of potential code reuse in new applications.

C. INHERITANCE

Inheritance uniquely distinguishes OO languages from other programming languages. It has even been called the only unique contribution of OO languages. (Korson and McGregor, 1990) Within the family of OO languages inheritance mechanisms vary widely. This section reviews the elasticity with which inheritance can be implemented, drawing out design implications for various inheritance strategies.

1. Definition

> *Inheritance enables the easy creation of objects that are almost like other objects with a few incremental changes. Inheritance reduces the need to specify redundant information and simplifies updating and modification, since information can be entered and changed in one place.* (Stefik and Bobrow, 1986, pg.41)

We adopt the view of Cook who defines inheritance as a composition mechanism that internalizes inherited attributes by late (execution time) binding of self-reference to the inheriting object. (Wegner and Zdonik, 1988, pg. 57)

Inheritance is here defined narrowly as a mechanism for resource sharing in hierarchies. (Wegner, 1987, pg. 169)

A subclass inherits all of the variables and methods defined for its superclass - regardless of whether those variables and methods were defined locally in the superclass or inherited from some other class. (Nelson, 1991, pg. 2)

Inheritance is a broad concept which serves multiple ends. Hence, inheritance must be approached from several perspectives to gain a fuller understanding of its conceptual diversity and utilitarian purposes.

First, inheritance is primarily a resource sharing mechanism, greatly extending reusability. The idea that opportunities for economy of design exist can be drawn from the observation that classes of objects exhibit conceptual, behavioral, or structural commonalities. Specifically, inheritance is a mechanism which permits the definition of one class to include the specification or implementation of another class on the basis of these commonalities.

Second, groups of classes can manifest collective commonalities which result in hierarchical relationships among the respective class definitions. Inheritance reifies these relationships into the actual implementation code. In the OO lexicon, an inheriting class is a subclass of the superclass from which it directly shares implementation code, and it is a descendent of all classes to which a path (from the subclass to higher levels in the hierarchy) can be traced. All the classes for which a given subclass is a descendent constitute the ancestors of that class.

Third, inheritance is a pliant concept. Depending upon the nature of the commonalities instigating the decision to share code, restrictions can be levied which shape the kinds of code sharing that are permissible. These formulations of the inheritance mechanism are discussed in the following subsections. It should be noted, however, that conceptual and programmatic difficulties often arise from language designs which emphasize but do not enforce particular inheritance restrictions.

Fourth, some OO languages implement single inheritance in which a subclass is only allowed to inherit from a single superclass while other OO languages implement multiple inheritance (MI) in which a subclass inherits from one or more superclasses. MI is a technique which powerfully increases the opportunities for code reuse. However, MI also introduces several complications, solutions for which are not uniform. These problems are discussed in the MI subsection below, and further examined in Chapter V.

Finally, inheritance mechanisms can undermine other desirable OO language features. Chapter V considers potential conflicts engendered by inheritance.

2. Inheritance Elasticity

As previously stated, classes provide instance variables, class variables, and methods, and they serve as templates for object instantiation. Inheritance entails decisions about the manner in which existing classes can be modified to form new templates. (Wegner and Zdonik, 1988) To facilitate discussion, inheritance can be described as a "...particular kind of incremental modification mechanism that transforms a parent entity P with a modifier M into a result entity R = P+M." (Wegner and Zdonik, 1988, pg. 55) P and M consist of sets of attributes (variables and methods) which may or may not be disjoint. Disjoint attribute sets do not present any particular problem. Problems arise in determining the manner in which overlapping attributes will be treated.

It bears emphasizing that inheritance is a subclassing, not a subtyping[9], mechanism. Inheritance realizes different kinds of templates depending on the constraints applied to the sharing of attributes.

[9]Subclassing is a set theoretical concept in which the members (variables and methods) of the subclass include all the members of the superclass. Subtyping is a behavioral concept in which any object of a subtype can be substituted for an object of the supertype and still respond to any service requests with the desired behavior.

a. *Logical Possibilities*

It is useful to classify the various logical possibilities for resource sharing under inheritance as either interface sharing or implementation sharing. An OO language may allow either or both of these forms of sharing, refining the individual categories through constraints.

Interface sharing entails the reuse of a class' interface, but not the actual implementation of the interface. Variable names and types are shared. Additionally, method names and parameters[10] are shared. Interface sharing can assume the following forms[11]:

- Variable names and types are shared.
- Method names and parameters are shared.
- Variable and method names, types, and parameters are shared.

Implementation sharing entails the sharing of method bodies. Such sharing offers the greatest opportunities

[10]Parameter sharing includes the names, number, and types of parameters as well as parameter qualifier distinctions (in versus out).

[11]Note that signature sharing is purely a syntactic matter. (Wegner and Zdonik, 1988) Specification sharing provides for sharing descriptions of the effects of methods. (Krakowiak et al, 1990) Hence, specification sharing allows for semantic associations. "In the current state of the art, the specification is only a comment and is not subject to any formal processing. However, it is considered an integral part of the type definition." (Krakowiak et al, 1990, pg. 13)

for code reuse. Implementation sharing can assume the following forms:

- Implementation code is directly shared.
- Implementation code is extended.
- Implementation code is overridden.
- Portions of implementation code are excluded.

Logically separate from inheritance, but an elemental consequence of the principles guiding the construction of class hierarchies, is the capacity to include new variables and methods in subclasses. Hence, inheritance takes shape as a cross-product of the listed options and extendibility. In most OO languages, inheritance combines both interface and implementation sharing. Virtual/deferred attributes allow for the inheritance of interfaces alone.

The following subsections draw upon Wegner and Zdonik's analysis of incremental modification in the context of inheritance. (Wegner and Zdonik, 1988) The authors assert that every class is a type describing a template. Their analysis is concerned with isolating the restrictions on inheritance (template modifications) that flow from different methods of specifying the behavior of types. The principle concern is that subtype behavior be compatible with supertype behavior. Different notions of compatibility emanate from the differing specification methods. There is a strong predisposition that a class hierarchy should be structured to account for substitution possibilities (of subtypes for supertypes). However, Wegner and Zdonik conclude that a strict interpretation of the subtype idea is overly narrow, intruding upon the flexibility which the subclassing mechanism permits. This leads inexorably to the conclusion that an OO language ought to provide weaker forms of typing/subtyping, thereby arming the programmer with the greatest leverage for designing class hierarchies. A supporting reason for such hierarchies is that objects in the real world do not often manifest relationships as conceived by strict subtyping, but exhibit a much richer set of similarities that class hierarchies should emulate.

a. Behavior Compatibility

Behavioral compatibility "...may be specified by algebras with a signature and a semantics." (Wegner and Zdonik, 1988, pg. 62) Hence, if classes are to be modeled such that the resulting class hierarchy doubles as a behaviorally compatible type hierarchy, then inheritance should be constrained to maintain a complete supertype/subtype relationship between superclass and subclass[12]. This entails some notion about the requirements for complete behavioral compatibility in these relationships. Specifically, the

[12]The critical notion is that the semantics of behavior must be compatible.

concept of substitutability must be defined for subtypes. Wegner and Zdonik define the principle of substitutability as follows:

> *An instance of a subtype can always be used in any context in which an instance of a supertype was expected.* (Wegner and Zdonik, 1988, pg. 65)

They then proceed to note that the only form of compatibility in which this notion of substitutability is preserved is that kind of inheritance in which subclasses are restricted to adding new variables or methods[13], and do not alter the semantics (modify variable, argument, or result domains) of superclass features. (Wegner and Zdonik, 1988) The sort of compatibility envisioned is therefore both syntactic and semantic. It is doubtful that a practical compiler could be designed to determine complete behavioral compatibility, especially as such compatibility cannot be specified in current programming languages.

Clearly, complete subtype compatibility is a highly restrictive notion and not enforced by current OO languages. In the analysis of class relationships, distinctions are often drawn between inheritance and type hierarchies. (Palsberg and Schwartzbach, 1990) Though many OOPLs identify subclassing with subtyping, it was previously noted that the two concepts are not the same. Consequently,

---

[13]Wegner and Zdonik use the term "horizontal extension." (Wegner and Zdonik, 1988, pg. 64).

it is possible to design type hierarchies which do not parallel the class hierarchy. Based upon this distinction, it is of interest to note that so long as data structures are encapsulated, objects of a class (as a data type) whose interface syntactically parallels the interface of other classes should be substitutable for objects of these other classes independent of inheritance relationships. This notion of subtype compatibility appears feasible. However, languages such as C++ and Eiffel restrict subtyping to inheritance relationships to simplify the complexity of algorithms performing compile time type checking, therefore improving performance.

b. Signature Compatibility

Signature compatibility (syntactic compatibility as described above) drops the requirement for behavioral compatibility. In particular, the domains of inherited attributes may be modified. The term "vertical modification" describes such domain changes. (Wegner and Zdonik, 1988, pg. 64) A signature compatible subtype (vertically modified) cannot be assigned to a supertype. A weaker form of substitutability is therefore offered in which an instance of a subtype can be used in read-only mode in any context a supertype is expected. (Wegner and Zdonik, 1988) The authors describe the relationships between entities in completely behavioral compatibility as consistent with 'is_a' hierarchies. However, as LaFonde and Pugh point out, 'is_a' is not the same as subtype, but rather is a specialization relationship. (LaLonde and Pugh, 1991) Such relationships better coincide with signature compatibility.

c. Name Compatibility

Name compatibility "...requires only the name and not the signature of the parent type to be preserved in the result." (Wegner and Zdonik, 1988, pg. 66) This is a simple and flexible form of incremental modification employed by many OO languages (such as Smalltalk). Name compatible modification entails searching the inheritance path (beginning with the result class) for the first occurrence of a name. Some OO languages modify the search algorithm by including syntax which permits definitions to be directly selected from ancestor classes (such as double dot notation in C++ or 'super' in Smalltalk).

d. Selective Inheritance

Selective inheritance[14] introduces the useful option of deleting inherited attributes. Selective inheritance, however, disrupts subtyping relationships should they exist. To facilitate reasoning about classes whose behavior is similar, but for which selective inheritance is employed, Wegner and Zdonik introduce the term "liketype."

---

[14]Wegner and Zdonik use the term "cancellation." (Wegner and Zdonik, 1988, pg. 67)

(Wegner and Zdonik, 1988, pg. 73) Liketypes logically include the other incremental modification mechanisms. Consequently, the authors recommend using like relationships to structure inheritance hierarchies, applying constraints to the like relationship as needed to achieve desired compatibility relationships between superclasses and subclasses.

Cancellation modification mechanisms alone provide for all the logical inheritance possibilities. It is interesting to observe that no OO languages that we know of implement cancellation mechanisms directly. The difficulties in managing intraclass linkages among attributes when cancellation is employed probably explains the absence of such cancellation mechanisms (as well as the adherence to some sort of strong typing philosophy).

3. Specialization

Several strategies can be employed to design class hierarchies. The possibilities include type, specialization, and like hierarchies. Additionally, classes may exhibit no abstract commonalties whatsoever other than code sharing or interface sharing. In the literature, however, specialization is typically described as the primary principle for hierarchy design. Yet, a precise formula for building such hierarchies has not found general acceptance. As the discussion in the previous subsection suggests, this probably reflects a desire on the part of designers to maintain maximum flexibility.

Specialization hierarchies are also called 'is_a' hierarchies (e.g., an eagle 'is_a' bird). Booch describes 'is_a' hierarchies as consisting of "...superclasses representing generalized abstractions, and subclasses representing specializations in which fields and methods from the superclass are added, modified, or even hidden." (Booch, 1991, pg. 56)

What qualifies as specialized behavior? What correspondence should there be between the mechanisms which implement inheritance and the abstractions which relate classes in a specialization hierarchy? A return to epistemological issues is evident. It appears reasonable that a standardized notion of specialization, based upon some sort of philosophical foundation, is required to introduce continuity to hierarchy construction and to facilitate the construction of compatible hierarchies (which, afterall, form the OO libraries central to code reusability).

Ultimately, the range of implementable hierarchies entails decisions about the distribution of responsibilities between programmers and language designers. Restrictive languages (basically strongly typed languages) ensure that programs are compiled in which undefined operations on objects are caught by the compiler. As flexibility increases, the programmer must ensure that undefined operations on objects do not happen (i.e., explicitly indicate to the system what class relationships prevail so that the compiler or run-time environment can enforce programmer intentions).

4. Multiple Inheritance

A subclass may inherit from several superclasses. Budd agrees that 'is_a' relationships should guide the construction of MI hierarchies (directed acyclic lattices), noting, however, that the resulting subclass should be viewed as a specialized "...combination or collection of several different components." (Budd, 1991, pg. 173) The idea of subclass as combination produces both the richness and difficulties that frame discussions about MI. In particular, what kinds of combinations should be permissible, and what status should be accorded subclass entities? Designers have not arrived at a consensus on these questions, which may explain why very few OO languages actually implement MI.

MI also introduces new problems. Prominent problems include name conflicts and inheritance from a common ancestor. Name conflict resolution strategies must be developed. Knudsen provides a useful framework for analyzing such conflicts, distinguishing horizontal[15] from vertical[16] name collision. (Knudsen, 1988) Such conflicts can be

[15]Attributes with the same name are inherited from multiple superclasses.

[16]Subclass possesses attributes with the same name as attributes in one or more superclasses.

characterized in three ways (Knudsen, 1988): (1) the same phenomena are defined; (2) casually related phenomena are defined; and, (3) unique phenomena are defined in which no collisions are permissible. The first method is handled by polymorphic techniques, the second by resolution operators[17] (such as double dot notation in C++), and the last will give rise to compile-time errors.

Inheritance from a common ancestor involves inheritance of attributes from superclasses whose inheritance paths converge at a common ancestor. At issue is the duplication of attributes. Should one or all attributes be inherited? If all are inherited, how are they to be distinguished? Solutions to this problem are discussed in Chapter V.

5. Delegation

Some OO languages approach reusability from a different philosophic perspective. In lieu of classes and inheritance to facilitate sharing the implementation of template abstractions, these languages "...directly use objects as prototypes from which the default behavior for concepts can be reused." (Lieberman, 1986, pg. 214) An object can delegate its attributes to one or more prototypes. Hence, an object receiving a message may defer to another object to

[17]Renaming or redefining subclass attributes is another solution. (Budd, 1991, pg. 174)

formulate the response. Delegation is the mechanism for implementing this in these OO languages. (Lieberman, 1986)

Proponents of delegation contend that it is more flexible and general than inheritance. Lieberman argues that inheritance fixes communication patterns between objects at instance creation time whereas delegation allows any object to serve as a prototype at any time. (Lieberman, 1986) However, delegation also carries the burden that objects are dependent on one another. Stein asserts that any changes to attributes, or their values, will affect both the object and the prototype. (Stein, 1987) More importantly, Stein presents a formal model which draws out the essential implications of classes qua templates: template instances are guaranteed to possess the same structural properties, but value independence. (Stein, 1987) The very flexibility of delegation eliminates any sort of structural guarantees and value independence for objects in an object hierarchy.

Delegation also raises epistemological questions. Given the run-time maneuverability of an object to delegate to other objects, it is perplexing as to what sort of knowledge is actually being modeled. Objects in a delegation hierarchy resemble amorphous entities amenable to the demands of the moment, but lacking assured structural or behavioral continuity.

In 1987, a compromise of sorts to the inheritance versus delegation 'controversy' was decided by the Treaty of Orlando. (OOPLSA Addendum to the Proceedings, 1987) Provisions of the treaty accepted that the object sharing mechanism could occur along three independent dimensions: (1) static or dynamic[18]; (2) implicit or explicit[19]; and, (3) per object or per group[20]. The position adopted in the treaty was "...that different programming situations call for different combinations of these features." (OOPLSA Addendum to the Proceedings, 1987, pg. 43) More than likely, the marketplace will be the final arbiter between the two approaches.

D. COMPOSITION

Another prominent relationship among real-world entities is composition (also called aggregation); complex objects can be conceived as consisting (i.e., being composed of) of aggregates of other objects. Hence, an object is 'part_of' another object (e.g., a wheel is 'part_of' a car). Stefik and Bobrow consider a composite object to be "...a group of

[18]The time that a system requires sharing patterns to be fixed (compile or runtime). (OOPSLA Addendum to the Proceedings, 1988)

[19]Sharing patterns can be declared by programmers (explicit) or automatically (implicit). (OOPSLA Addendum to the Proceedings, 1987)

[20]Sharing can be specified for an object at a time (per object) or for a group of objects at a time (per group). (OOPSLA Addendum to the Proceedings, 1987)

interconnected objects that are instantiated together, a recursive extension of the notion of object." (Stefik and Bobrow, 1986, pg. 51) Several ideas can be drawn from this conception of composite objects.

First, composition is another mechanism for reusability. The class template for a group of objects may include the previously defined templates for other classes of objects - redefinition is not necessary. Booch notes that such composition relationships can be implemented through two mechanisms (Booch, 1991): (1) declaration of class instance variables as user defined types; and, (2) declaration of formal parameters for class methods as user defined types (as a parameter to the class interface).

Second, the interconnectedness of objects in composition relationships occurs through the respective object interfaces. This serves to preserve encapsulation. Nevertheless, the interconnectedness also establishes a coupling between respective classes. The implications of this coupling will be examined in Chapter V.

Third, composition should not be confused with inheritance (either single or multiple). In particular, Halbert and O'Brien point out that a "...subclass inherits from a superclass only once while aggregation allows more than one instance of a particular object type." (Halbert and O'Brien, 1987, pg. 76)

Fourth, the notion of composition as a recursive definition highlights the fact that members of a composite object may themselves also be composite objects. Consequently, any level of nested complexity is possible.

Finally, it is interesting to ask how composite objects differ from collections of objects cooperating collectively to achieve a systematic pattern of behavior. The answer is that the differences are mostly in the respective degrees of abstraction and complexity. Composite objects can themselves be viewed as systems (e.g., a car). However, the level of complexity and abstractness for systems such as a factory is elevated enough that it should not be localized into a single object. These application specific decisions reflect design considerations about the distribution of knowledge, and visibility of objects.

E. POLYMORPHISM

Polymorphism is one of the more abstruse concepts in the OO literature. Consequently, a variety of approaches are taken toward delimiting its meaning. That inheritance, specialization, message passing, and polymorphism all interact to achieve reusability and extendibility further complicates isolating the content and effects of polymorphism.

Budd observes that definitions of polymorphism often overlap other concepts such as overloading. (Budd,1991) This section reviews several polymorphic mechanisms without regard to delineating the boundaries with overlapping concepts. The intent is to establish the manner in which OO languages implement polymorphism.

1. Abstract Qualities

In programming languages, "a polymorphic object is an entity, such as a variable or function argument, that is permitted to hold values of differing types during the course of execution." (Budd, 1991, pg. 185) Most OOPLs provide an efficient message passing construct that enables receivers of messages to change. (Ingalls, 1986) Finally, Meyer states that in strongly typed environments (such as C++, Eiffel, etc.), the changing among types or message receivers is constrained by inheritance. (Meyer, 1988)

What emerges is the notion that polymorphism describes a group of mechanisms that permit programming constructs (i.e., method names, method arguments, and objects) to shift definitions in the course of program execution. Individual languages must be studied to understand how the shifting is accomplished. For example, some languages distinguish the static, declared class of an object from the dynamic class of its value. (Meyer, 1988) Polymorphism is managed in these languages through manipulations of references and pointers. Other languages manage polymorphism by binding values to objects at run-time only.

2. Polymorphic Names

This form of polymorphism occurs when the same message can be sent to different objects. It is commonly associated with the overloading of function names. Hence, several (possibly unrelated) classes may have a method with the same name. A standard example is the method print[21].

3. Polymorphic Names and Arguments

Another variant of overloading occurs when methods with the same name have different argument cardinality or different argument types[22]. The methods are all grouped within a single class. A standard example is the constructor function in C++ classes.

---

[21]For example, several classes may have a method named *print* which has no arguments. Individual objects from the different classes, when receiving the *print* request, understand that the local implementation of the *print* method is to be used. Polymorphism, used in this manner, avoids the undesirable construction of large case statements which match methods to objects. Such large case statements also assume too much knowledge on the part of one object about other objects.

[22]Micallef describes this as "multiple polymorphism." (Micallef, 1988, pg. 32) in which there is more than one polymorphic variable. She distinguishes this from simple polymorphism in which the "operation invoked is dependent on the type of only one argument, the receiver of the object." (Micallef, 1988, pg. 32) The idea is that the same method name may be employed by different classes (simple polymorphism), or by more than one method within a single class (multiple polymorphism).

4. Varia

The forms of polymorphism listed above represent the basic cases found in most OO languages. Budd additionally includes overriding, virtual, deferred, and parametric techniques[23] among his list of polymorphic mechanisms. (Budd, 1991)

F. CONCLUSIONS

The OO paradigm has evolved since the introduction of the first OO language, CEMBALO (Meyer, 1988), in 1968 to encompass objects, classes, and inheritance. It is inheritance (or delegation) which uniquely distinguishes OO languages from other programming languages. Languages which include objects and classes, but not inheritance/delegation are called object-based languages (for example, Ada). Currently, the greatest impediment to the commercial ascendancy of the OO paradigm as the methodology of choice for language design is the lack of conceptual standardization.

[23]Overriding occurs when a subclass redefines the body of an ancestor method. The other techniques are discussed in the section describing classes.

III. OO DEVELOPMENT

A. SOFTWARE ENGINEERING METHODOLOGIES

Software engineering is the application of science and mathematics to the problem of making computers useful to people via software. (Berzins and Luqi, 1991, pg. 1)

Software engineering finds its genesis in the perception during the 1960's that software production was a disorganized process, the vagaries of which often resulted in avoidable increases in the total cost of software over the lifetime of a product[1]. (Schach, 1990) Computer scientists set about on a scientific search for principles which would objectify the process of software development. As the discipline evolved, many of the software properties discussed in Chapter I were established. The search for development methods which accentuated these properties has naturally been influenced by the underlying philosophy (or paradigm) adopted for understanding application domains.

Currently, many computer scientists are investigating strategies for managing a transition from non-OO based

[1]The manifold problems which produced huge increases in the total cost of software during the 1960's have collectively been termed the "software crisis." (Schach, 1990, pg. 5) Particular emphasis has been directed toward the excessive costs associated with software maintenance. (Booch, 1987)

development methodologies to OO based methodologies[2]. A wide variety of opinion exists as to the preferred course to follow. This chapter reviews software development aspects programmers should consider in assessing the relative merits of various strategies, considers several approaches to OO development, and advances development recommendations.

1. Lifecycle Organization

In object-oriented analysis, we seek to model the world by identifying the classes and objects that form the vocabulary of the problem domain, and in object-oriented design, we invent the abstractions and mechanisms that provide the behavior that this model requires. (Booch, 1991, pg. 141)

The advent of software engineerin produced a mindset which focused on decomposing complex ideas and processes into simpler ones. Decomposition of the development process itself produced models of varying constitution, but models usually included the following stages[3]: requirements analysis, specification, design, implementation, maintenance, and retirement. (Schach, 1990) Structured techniques based upon

[2]For example, Booch 1991, Coad 1991, Li 1991, Pun 1991, Odell 1991, and Arnold 1991 all discuss new approaches to software development based upon the OO paradigm. Debate focuses on the most efficient manner in which to shift from current methodologies based upon structured techniques (analysis, design, and implementation guided by functional decomposition of the application domain) to OO techniques. Advocacy ranges from evolutionary to revolutionary strategies. (Li, 1991)

[3]These stages are collectively referred to as the "software lifecycle." (Schach, 1990, pg. 43)

functional decomposition were used to further simplify each stage in the software lifecycle.

The OO development process is generally structured in terms of OOA, OOD, and OOP. OOA is concerned with defining concepts in the problem domain. It is through OOA that knowledge about the real-world is captured[4]. OOD extends the results of OOA, uncovering entities missed by OOA, others that meet user requirements, and still other entities that are needed to consolidate an application into a serviceable tool (for example, user interfaces and task managers). OOP involves the actual implementation of the results of OOD. There is no rigid formula for conducting these stages; various temporal schemes can be utilized, the net results of which are iterative development processes best described as "...round-trip gestalt." (Booch, 1991, pg. 188)

Debate exists over "...whether to replace structured techniques and functional decomposition by object-oriented techniques, or whether to look for a pragmatic solution in which existing investments are retained to a significant degree and tools and methods modified to encompass the object-oriented paradigm[5]." (Henderson-Sellers and Constantine, 1991,

---

[4]Hence, as suggested in Chapter II, OOA requires subject matter expertise.

[5]Henderson-Sellers and Constantine note that the OOA/OOD/OOP breakdown can be handled in any of several ways. That is, though the objective of each stage is to produce OO

pg. 18) Consequently, strategies for OO development entail decisions about which techniques to apply during each lifecycle stage. Ultimately, where an analyst falls on this issue largely depends upon his philosophical predisposition to adhere to purely OO concepts and techniques, and upon the availability of OO development environments[6]. Currently, the preponderance of development strategies employ modified structured techniques[7].

It should be pointed out that OOA/OOD concepts and techniques are serviceable tools for development leading to implementation in non-OOPLs. The OO approach to knowledge representation (objects and their relationships) facilitates problem understanding in a manner that is transferable to non-

---

results, the techniques adopted at a given stage may be purely OO, or they may be structured techniques modified to produce OO usable results. (Henderson-Sellers, 1991) The critical question is whether structured techniques can in fact be facilely modified to accommodate OO thinking. The answer to this question entails both conceptual and economic considerations: (1) can techniques based upon functional decomposition be used to uncover fundamentally different entities and relationships (i.e., objects, classes, and inheritance); and, (2) can structured techniques, if used, be employed efficiently with a minimum of modification.

[6]It is one thing to argue for purely OO techniques, quite another to actually provide such methods and tools.

[7]Data flow diagrams (DFD), entity relationship diagrams (ER), state transition diagrams, or event-response diagrams are incorporated into many OO development approaches. For example, Booch 1991, Rumbaugh 1991, and Li 1991 all include some of these techniques as part of their development methodology.

OO programming. Specifically, OOA/OOD can serve to structure high-level abstractions which can then be tailored to suit the requirements of individual programming languages.

2. Conceptual Organization

When investigating proposed approaches, it is important to consider the primary conceptual blocks (or models) used to abstract a problem: does an approach directly compose analysis in terms of objects, classes, and hierarchies? Many structured techniques do not abstract problem entities in this manner. In particular, structured techniques map real-world entities to functions and data. (de Champeaux et al, 1990, pg. 135 - 139) Heuristics must then be applied to transition to an OO conceptualization.

Two other difficult development problems must also be investigated. First, OO development requires methods for recognizing and structuring systems. For present purposes, systems analysis in the OO framework is narrowly conceived of as a process which determines groupings of objects that accomplish some pattern (or subpattern) of collective behavior. At issue is how analysts go about identifying and relating these abstractions. The problem is a subtle one since the system behavior, exercised as collections of cooperating objects, is diffused throughout the class hierarchies. Analysts are therefore confronted with a twofold problem: (1) determine the functional responsibilities of a system and its constituent subsystems; and, (2) determine the best manner for distributing these behaviors among the classes. Few proposed OO development approaches adequately handle this analytical problem; a problem, it should be noted, that structured techniques manage quite well.

Second, reusability inevitably covers the entire development spectrum. Viewed from this perspective, it is appropriate to question how development should be conducted given that analyses, designs, and programs can be reused as elements in future development efforts. Specifically, OOA, OOD, and OOP no longer focus solely on the present project, but potentially supply source material for future projects. What concepts should guide development under these circumstances? Are some designs more reusable than others? Can potential reusability be measured? Clearly, reusability requires further research.

3. Notational Organization

Closely allied to conceptual organization is the notational scheme adopted. Are the transitions between development phases enhanced or impeded by notational tools? Does the notation employed completely and consistently describe the models being used for understanding a problem (Arnold et al, 1991)? In particular, is a consistent representation utilized? It has been argued that representational shifts have stymied developers as they move from analysis to design. (Coad and Yourdon, 1991) A solution is arrived at "by applying a uniform underlying representation for organizing data and its exclusive processing - that of Classes and Objects within those Classes...." (Coad and Yourdon, 1991, pg. 21) Again, approaches vary according to the philosophic adherence to pure OO concepts and techniques.

B. OO DEVELOPMENT APPROACHES

OO practitioners are in search of methods whose logic is infused from the start by OO constructs and objectives. Given the lack of standardization in the field, it is not surprising that there exist wide differences in OO development approaches. The positions advanced by some of the better-known OO advocates will now be reviewed.

1. Coad/Yourdon

In apprehending the real world, men [people] constantly employ three methods of organization, which pervades all their thinking. (Coad and Yourdon, 1991, pg. 1)

Coad and Yourdon proceed on the assumption that analysis/design thinking should parallel the patterns by which people ordinarily organize knowledge. The three methods of organizing knowledge are (Coad and Yourdon, 1991): (1) objects and their attributes; (2) distinctions between objects and component parts; and, (3) distinctions between classes of objects. This knowledge is garnered in OOA through five activities (Coad and Yourdon, 1991): (1) finding class-&-objects; (2) identifying structures; (3) identifying subjects; (4) defining attributes; and (5) defining services[8]. The activities can be pursued in any order, and generally move from higher to lower levels of abstraction.

OOD, in the Coad/Yourdon approach, takes the results of OOA and further refines the organization of knowledge. Additionally, specific requirements of the application are introduced by organizing design into four components (Coad and Yourdon, 1991): (1) the problem component, which models the real-world problem space; (2) the human interaction component, which models how a human will command system and how a system will present information; (3) the task management component, which addresses concurrency control; and, (4) the data management component, which provides the infrastructure for the storage and retrieval of objects from a data management system. OOA results form the bulk of the problem domain component.

As analysis and design is refined, particular emphasis should be applied to reducing connections between objects and

---

[8]Class-&-objects assume their usual meaning, structures include generalization-specialization and whole-part structures, subjects are mechanisms for guiding analysts/experts through complex models, attributes are data maintained about the state of an object, and services are behaviors objects are responsible for. (Coad and Yourdon, 1991)

between classes. (Coad and Yourdon, 1991) Specifically, they recommend controlling the following forms of coupling (Coad and Yourdon, 1991): (1) interaction coupling (limiting parameters in messages to three or fewer, and simplifying the number of messages sent and received by individual objects); and, (2) inheritance coupling (maximizing superclass/subclass connections along generalization-specialization lines). The authors also recommend that cohesion be maximized as follows: (1) services should carry out one function; (2) classes should contain no extra attributes or services; and, (3) inheritance should portray specialization cohesion, not arbitrary relationships.

Though great emphasis is placed upon the use of a unifying OO notation, comparatively little advice is directed toward actual development tools. The authors do advocate utilizing a CASE tool for OOA, and also recommend using summary cards for manually conducting analysis and design. (Coad and Yourdon, 1991)

2. Booch

> *Object-oriented design is not a process that starts with a requirements specification, ends with a blueprint for implementation, and requires a miracle somewhere in between. We suggest that it allow an evolutionary development, a view consistent with Boehm's spiral model of software development.* (Booch, 1991, pg. 190)

We believe that Booch's discussion on OOA and OOD (Booch, 1991) is as much a diatribe on OO philosophy as it is on OO techniques. Particular care is taken to emphasize that the OO approach to software development is a voyage of discovery and invention for which there are no hard and fast rules. Consequently, considerable effort is spent explaining the limited applicability of structured techniques and waterfall lifecycle development. Structured techniques tend to reflect a bias toward algorithmic decomposition inappropriate to real-world modeling of interacting objects, and the waterfall lifecycle is a "...fundamentally poor process, and generally violates many of the principles of sound engineering practice." (Booch, 1991, pg. 189)

The foundation of the Booch approach is the isolation and iterative refinement of problem abstractions. He directly confesses that OO development is a fuzzy process in which domain expertise, experience, and intuition all play a role in uncovering relevant abstractions at appropriate levels of detail. In describing this evolutionary process, he highlights four prominent activities and illustrates several techniques, as described in the following subsections.

Development, for Booch, focuses on defining three principal constructs: objects, classes, and mechanisms. The following subsections review Booch's suggested techniques and organizing activities.

a. Techniques/Notation

Booch underlines the need for taking multiple views on complex systems. (Booch, 1991) Hence, he advocates the use of the following diagrams: class, object, module, and process. The first two diagrams describe the logical view of a system while the last two describe the physical structure of a system. These diagrams capture static semantics. Dynamic system properties are captured in state transition diagrams and timing diagrams. Collectively, these techniques preserve the knowledge garnered during the four organizing activities.

Class diagrams indicate class relationships[9], class utilities[10], class categories[11], superclasses, fields, and operations. State transition diagrams show the state space of a class - events causing state transitions. Object diagrams "...show the existence of objects and their relationships in the logical design of a system, and illustrate the semantics of key mechanisms in the logical design." (Booch, 1991, pg. 169) Hence, object diagrams are

---

[9]Class relationships include inheritance, instantiation, using, and metaclass relationships. (Booch, 1991)

[10]Class utilities are free subprograms. (Booch, 1991) That is, they are operations which are not meaningfully encapsulated by any particular object. Instead, these operations are grouped into utility classes from which they are accessible, but cannot be redefined. (Booch, 1991)

[11]Class categories are logical collections of classes. Each class within a category has an associated visibility: private to the category, externally visible, or imported from another category. (Booch, 1991)

used to depict object properties, relationships, visibility, and message synchronization. Timing diagrams indicate the flow of control among collaborating objects. Module diagrams show the allocation of classes and objects to modules, and module visibility. Booch uses subsystems to group logically related modules. Finally, process diagrams describe processor allocation for applications with concurrent tasks.

b. Activities

Booch maintains that four activities typify OO development (Booch, 1991): (1) identify classes and objects at a given level of abstraction; (2) identify the semantics of classes and objects; (3) identify relationships among classes and objects; and, (4) implement classes and objects. The first activity involves "...the discovery of key abstractions in the problem space and the invention of important mechanisms that provide the behavior required of objects that work together." (Booch, 1991, pg. 191) The second activity "...establishes the meanings of classes and objects, viewing each class from the perspective of its interface." (Booch, 1991, pg. 192) The third activity establishes "...how things interact within the system." (Booch, 1991, pg. 193) Finally, the fourth activity involves "...design decisions concerning the representation of the classes and objects we have invented, and allocating classes and objects to modules, and programs to processors." (Booch, 1991, pg. 195)

3. Wirfs-Brock

Model your design as clients and servers who collaborate in ways specified by contracts.
(Wirfs-Brock, Wilkerson, and Wiener, 1990, pg. 32)

The Wirfs-Brock approach to design squarely focuses on maximizing and preserving encapsulation. (Wirfs-Brock and Wilkerson, 1989) The client-server concept moves analysis toward a responsibility-driven, contract perspective on entity interactions which forces analysis and design away from implementation/structural details and closer to behavioral abstraction. (Wirfs-Brock and Wilkerson, 1989)

Analysis in the exploratory phase of system design moves in the following directions (Wirfs-Brock, Wilkerson, and Wiener, 1990): (1) find objects; (2) determine object responsibilities; and, (3) determine object collaborations[12]. Heuristics and guidelines are offered to conduct the process. It is recommended that class cards be maintained to record information about classes, class responsibilities, and collaborations.

Wirfs-Brock proposes that the next phase of design focus on structuring inheritance hierarchies using hierarchy graphs, Venn diagrams, and contract analysis. (Wirfs-Brock,

---

[12]Collaborations entail class interactions. Such interactions are uncovered by analyzing class communication paths, particularly is_part_of, has_knowledge_of, and depends_upon communication. (Wirfs-Brock, Wilkerson, and Wiener, 1990)

Wilkerson, and Wiener, 1990) During this phase many distinctions are drawn out (Wirfs-Brock, 1990): (1) abstract and concrete classes are determined; (2) 'kind_of' hierarchies are built in which common responsibilities are moved up the hierarchy, abstract classes are added, and unnecessary[13] classes are eliminated; (3) contract analysis directs the reassignment of responsibilities, and the uncovering of new responsibilities; (4) class cohesiveness is maximized; and, (5) the number of class contracts is minimized. The overriding objective is that "...each class have a single, overarching purpose; each class should serve one main function in the system of which it is a part." (Wirfs-Brock, Wilkerson, and Wiener, 1990, pg. 121)

Finally, and most interestingly, collaboration graphs and subsystem cards are employed to streamline collaborations among classes[14]. (Wirfs-Brock, Wilkerson, and Wiener, 1990) Again, heuristics and guidelines (emphasizing collaboration analysis) are offered to assist in identifying subsystems. Proposals include the following (Wirfs-Brock, Wilkerson, and Wiener, 1990): (1) drawing collaboration graphs; (2)

---

[13]Unnecessary classes are those which do not add functionality. (Wirfs-Brock, Wilkerson, and Wiener, 1990)

[14]Collaboration graphs describes communication paths among classes, and subsystem cards describe a subsystem's responsibilities (contracts) and the class to which the contract is delegated. (Wirfs-Brock, Wilkerson, and Wiener, 1990)

determining strongly coupled classes (including transitively coupled classes); (3) simplifying and minimizing interactions; (4) minimizing subsystem responsibilities delegated to a class; and, (5) minimizing contracts supported by a subsystem. The general idea is to efficiently distribute responsibilities throughout the hierarchies on the basis of contract considerations stimulated by subsystem analysis.

C. RECOMMENDED OO ANALYSIS AND DESIGN METHODS

The OO methodology for representing knowledge about real-world objects is comparatively straightforward. However, determining the relevant objects and their systematic relationships is difficult. A particular sticking point is the lack of strategic thinking on systematic organization. This is, perhaps, unavoidable for a methodology that selects objects and not processes as the analytic ambit.

There are no rigid formulas for conducting OOA and OOD. As demonstrated by the various approaches discussed in the previous section, these steps iteratively inform and improve one another. However, certain themes can be culled which can be applied as a 'backbone' upon which tailored modifications can be extended. These themes can be thought of as recommendations for organizing OOA and OOD:

- In a concession to structured design, functionally decompose a system into its major constituent subsystems. This decomposition should serve as a checklist against which the actual evolving design can be assessed to ensure that principle subsystem responsibilities are accounted for.

- Complement system analysis with a parallel analysis of supporting application requirements - concurrency control, interface, and database management. Maximize reuse of previously designed application code. When possible, allow application support objects to perform subsystem responsibilities[15].

- Analyze principle subsystem responsibilities.

- Identify problem domain objects/classes. Subject experts, and prior designs should be exploited to the maximum extent possible.

- Analyze principle object responsibilities. Note collaborations among objects[16].

- Analyze object variables. Decide state information that each object needs to preserve to fulfill responsibilities. Account for variables shared by objects, variables that can be calculated, and variables that can be decomposed[17]. (dePaula and Nelson, 1991).

- Group objects into subsystems. Note that an object may participate in more than one subsystem.

- Check to ensure that cooperating objects account for behavior expected from respective subsystems. Adjust object collaborations to achieve efficient distribution of system responsibilities: promote tight object/class cohesion, minimize object/class

---

[15]Integrating problem domain responsibilities into application object responsibilities is one of the more subtle and difficult tasks faced by the analyst/designer.

[16]Analysis should allow for decisions already embedded in available libraries.

[17]Variable decomposition takes account of composition relationships. (dePaula and Nelson, 1991)

linkages within a subsystem, and reduce the number of message parameters where possible[18].

- Group classes into specialization hierarchies[19]. This presupposes a decision about the semantics which will control the formulation of specialization relationships.

- Streamline class hierarchies. This includes segregating abstract from concrete classes[20], factoring common methods as high as possible, and eliminating unnecessary classes. (dePaula and Nelson, 1991) Note that common protocol design will include decisions about virtual and pure virtual features.

Tools for assisting analysis/design can be located in the various approaches previously listed. CASE tools created specifically for OO purposes should be used when available. However, when considering CASE tools, many of the issues discussed in Section A need to be carefully assessed.

These recommendations constitute a starting point. An analyst must first ensure that a strong foundation in OO techniques and philosophy has been acquired - knowledge which extends beyond mere facility with a particular pure or hybrid

---

[18]An object overburdened with subsystem responsibilities suggests that the object/class should be decomposed into smaller, more specialized objects/classes. This will assist in promoting object/class cohesion.

[19]Note that hierarchy construction requires prior consideration of OOL selection - single or multiple inheritance strategies must be decided. Also, multiple inheritance strategies must account for resolutions to conflicts discussed in Chapter II.

[20]Abstract classes should formulate a common protocol. Additionally, abstract classes should not inherit from concrete classes. (dePaula and Nelson, 1991)

OOPL. This foundation will be needed to resolve many of the OO mechanism conflicts which are addressed in Chapter V.

IV. FUNCTIONAL DECOMPOSITION AND SUBCLASS RESPONSIBILITY

A. FUNCTIONAL DECOMPOSITION

Chapter III considered the applicability of structured design techniques to OO problems. Although the emphasis of structured design falls squarely upon procedures and not objects, many of the concerns which prompted structured design in the first place (modularity, flexibility, reliability, cohesion, coupling) receive corresponding importance in OOP. Hence, it is not unreasonable to expect that some of the arguments, if not techniques, of structured design applies to OOP. This section addresses one area which merits attention: functional decomposition.

1. Functional Decomposition and Subclass Responsibility

It has been noted that the pure OO methodological framework "...does not totally neglect structured tools and experience; rather, it defers it to a more detailed design level." (Henderson-Sellers and Constantine, 1991, pg. 14) Specifically, design of methods is "...essentially identical to structured, functional decomposition as developed over the last twenty years or so." (Henderson-Sellers and Constantine, 1991, pg. 14) Observe that "this does not contradict the object-oriented paradigm since at this level the implementation of the features is hidden and changes in the implementation therefore have, at least in principle, no repercussions on the rest of the software system." (Henderson-Sellers and Constantine, 1991, pg. 14)

Several software qualities and heuristics have been identified as conducive to better structured designs. (Yourdon and Constantine, 1978) In the context of structured design, a module is identified with a single functional purpose. The unifying purpose varies according to the organizing strategy adopted (for example, transaction analysis or transform analysis). Modules are decomposed (program structures are organized) by assessing, among other things, factoring[1], cohesion[2], and coupling[3]. Given the identification of modules with functional purpose, this resolves into a process of functional decomposition.

Cohesion, coupling, and factoring are also relevant to the design of methods. Subclass responsibility was previously

---

[1]Factoring is a term which describes the degree to which control and coordination functions are performed by higher-level modules in a hierarchy (produced by modular decomposition), and processing is delegated to subordinant modules. (Yourdon and Constantine, 1978).

[2]Cohesion reflects a semantic or procedural unity exhibited by statements which suggests bundling into a single module is appropriate.

[3]Coupling is a measure of the degree to which separate program elements are independent of one another. (Yourdon and Constantine, 1978)

defined (Chapter II) as a situation in which ancestor class methods rely on the knowledge that descendants must implement certain methods. As such, a design providing for subclass responsibility can be viewed as a subset of functional decomposition.

Figure 1 can be used to illustrate subclass responsibility. Superclass A contains methods 1 and 2 while subclasses B, C, and D contain methods 3 and 4, 5 and 6, and 3 and 7 respectively. Method 1 invokes method 3, but Class A does not implement it. This situation therefore requires that Class A be implemented as an abstract class, or that an instance of Class A cannot call method 1. The invocation of method 3 takes the form ***self->3***. Observe that subclasses B and D implement method 3, but that subclass C does not. Thus, Class C, like Class A, must either be an abstract class or **instances of Class C cannot call method 1. Finally, methods** 4, 6, and 7 simply indicate that other methods may be

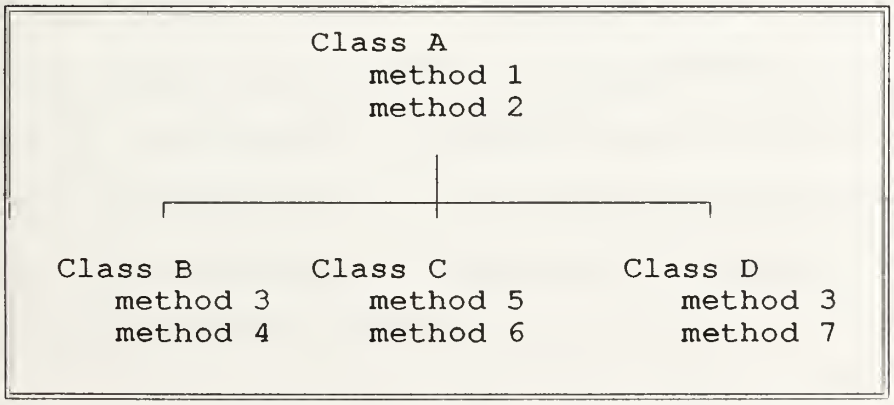

Figure 1: Subclass Responsibility

implemented by subclasses. In this example, these other methods do not interact with method 3 or method 1. The following subsections review problems associated with subclass responsibility and use of cohesion, coupling, and factoring to reduce these problems.

a. Subclass Responsibility

Subclass responsibility does not fit easily into the design recommendations listed in Chapter III. First, as illustrated in Figure 1, it is questionable whether subclasses should be allowed to exclude methods assumed to be implemented. This may be possible for liketype systems as described in Chapter II, but should not be attempted in dynamically typed languages. A class may legally inherit, but not implement a pure virtual method. In a strongly typed language such as C++, such classes are automatically recognized as abstract classes. Hence, run-time errors will not occur since objects from these classes cannot be instantiated[4]. No such protective mechanisms are available in dynamically typed languages like Smalltalk. Consequently, the onus is shifted to the designer to ensure that all possible avenues for arriving at such an invocation are precluded.

Second, subclass responsibility clearly establishes an undesirable coupling between ancestor and

[4]It is assumed the design recommendation that abstract classes not inherit from concrete classes is also observed.

descendent classes. It requires a cooperative design effort between respective class designers. Notice that it also requires that descendent classes have some awareness of ancestor class implementations (i.e., method 3 occurs in the specific context of method 1).

Third, in languages such as C++, method 3 must be declared (although not defined) in Class A. Why not supply default behavior, perhaps an error message, to protect designers of descendent classes? In other words, employ virtual functions which can be overridden by descendent classes.

Fourth, it was asserted in Chapter III that abstract classes should provide a common protocol and define common behavior for descendent classes. Again, if subclass responsible behaviors (for example, method 3) are not applicable to all concrete descendants (for example, Class C), they should not, under this formulation, be designed into the concerned abstract class.

Finally, designers must carefully consider the visibility of subclass responsible behaviors. Given the linkages (discussed below) established by subclass responsible design, it is questionable whether such behaviors should form part of the external interface.

The preceding discussion leads to the following recommendations when employing subclass responsible designs:

(1) classes such as Class A in Figure 1 should be abstract classes; (2) every concrete descendent class should implement[5] the subclass responsible behavior; and, (3) carefully consider whether to include abstract classes which inherit from concrete classes implementing subclass responsible behaviors.

b. Design Heuristics for Subclass Responsibility

Should designers elect to organize behavior using subclass responsibility, the techniques of structured design can be used to structure solutions. The starting point is a class with one or more methods which are excessively large[6], or not conceptually unified. Structured techniques can then be applied to decompose these methods.

(1) Cohesion. "Cohesion is the measure of the strength of functional relatedness of elements within a module." (Page-Jones, 1988, pg. 83) Modular elements[7] are related (or associated) by virtue of some property they have in common. (Yourdon and Constantine, 1978) Included among

---

[5]Either directly or through inheritance from another concrete class.

[6]The computer science literature is generally ambiguous about what constitutes an overly large module since the magnitude is influenced by notions of cohesion. However, half a page, about 30 lines of program statements in a high level language, has been offered as tolerable. (Page-Jones, 1988)

[7]In the present context, modules refer to methods and elements refer to statements or groups of statements in a method.

these associative properties are the following: functional[8], sequential[9], communicational[10], procedural[11], temporal[12], logical[13], and coincidental[14].

In the context of a class, a method is associated with a single form of behavior. Hence, functional cohesion should determine whether elements are bundled into a single method. Although sequential and communicational cohesion have also been supported as reasons for bundling elements into a single module (Page-Jones, 1988), these are data-oriented associations which are incompatible with the behavioral underpinning to OO methods. In passing, it should be noted

---

[8]Functional cohesion relates program elements that all contribute to the accomplishment of a single problem-related task. (Page-Jones, 1988)

[9]Sequential cohesion involves activity such that output from one activity serves as input to the next activity. (Page-Jones, 1988)

[10]Communicational cohesion relates elements which all share the same input, or contribute to the same output. (Page-Jones, 1988)

[11]Procedural cohesion relates activities associated by control flow. (Page-Jones, 1988)

[12]Temporal cohesion involves activities related in time. (Page-Jones, 1988)

[13]Logical cohesion relates activities of the same general category (for example, means of transport), the execution of which is determined from outside the module. (Page-Jones, 1988)

[14]Coincidental cohesion relates activities with no meaningful relationship to one another. (Page-Jones, 1988)

that the separate behaviors which are combined by these two forms of cohesion are not available to other methods or to descendent classes[15]. The other forms of cohesion represent looser associations which should not be used to build methods. Consequently, elements that cannot be tied together through functional cohesion should be broken out as distinct methods.

The problem remains, however, of elements within a functionally cohesive method which represent pieces of behavior which are conceptually the same[16], but which require different implementations depending upon the objects to which they are applied. These statements, not surprisingly, can be broken out using subclass responsibility

(2) Coupling. Implicit in the notion of cohesion is the idea that large, uncohesive modules should be partitioned into smaller, conceptually unified modules. (Page-Jones, 1988) "It is vital that this partitioning should be carried out in a way that the modules are as independent as possible - this is the criterion of coupling...." (Page-Jones, 1988, pg. 57) Coupling in the OO environment has so far been described as a linkage established between two

[15]These behaviors can be made available by duplicating definitions (i.e., defining methods which implement the same behavior).

[16]Yourdon and Constantine describe these as "processing elements" in distinction to instructions or statements. (Yourdon and Constantine, 1978, pg. 97)

classes on the basis of the knowledge possessed by one class of the other's external and internal interfaces. Chapter V/Section A analyzes other forms of OO coupling, discussing aberrant forms of coupling in which classes possess direct knowledge of implementation details.

It was previously suggested that subclass responsibility creates an undesirable coupling by requiring subclass designers to understand in what contexts subclass responsible behaviors are invoked. This is evidenced by the fact that such behaviors have a specific role to play in completing the behavior expected of the calling method, and generally are not designed to fulfill independent behavioral duties (i.e., subclass responsible methods approximate what were termed auxiliary methods in Chapter II).

Structured techniques usefully define two forms of coupling that should be avoided or minimized when designing subclass responsible relationships. First, data coupling should be minimized. Data coupling is a linkage achieved through parameter passing. At issue is how many parameters are passed, what details are revealed by the parameters, and how the parameters are subsequently used (side effects). Designers should avoid passing large numbers of parameters, and should preserve encapsulation of structural details. Second, control coupling entails the passing of information intended to control the internal logic of the receiving module. Control coupling requires that the calling method have knowledge about the details of subclass responsible behaviors. This has obvious implications for modifications effected on subclass responsible behaviors and should be avoided. Moreover, these complexities multiply as fan out[17] increases.

(3) Factoring. "Factoring is the separation of a function contained as code in one module into a new module of its own" (Page-Jones, 1988, pg. 103) It is used to achieve one or more of the following (Page-Jones, 1988): reduce module size, achieve top-down design, avoid function duplication, separate work from management, generalize modules, simplify implementation. Note that factoring includes more than subclass responsibility. Hence, once a reason for pursuing factoring has been selected (reduce module size, generalize modules, etc.), structured techniques such as DFDs and structure charts can be used to examine methods and determine the merits of alternatives.

Several issues need to be clarified, however, before factoring is attempted. First, a rationale must be

[17]Fan out is a magnitude describing the number of modules subordinate to a higher level module. In the present context, fan out describes the number of descendent classes defining subclass responsible behavior for a particular invocation. Note that difficulties are even greater if more than one method in an ancestor class contains calls to subclass responsible behaviors.

established. Top-down design of methods in a class is probably not a cogent reason for decomposition. Module size, of itself, is not a sufficient reason for breaking out behavior (i.e., large does not necessarily mean uncohesive). On the other hand, avoiding function duplication and generalizing methods are good reasons for decomposing methods.

Second, once decomposition has been accomplished, decisions based upon OO considerations must be made about what to do with the results. Again, designers return to the requirement for formulating a methodology for allocating knowledge among classes. Matching behavior to objects in the problem has often been advanced in this thesis as one such criterion (i.e., responsibility driven analysis). Nevertheless, this rather facile solution requires substantial amplification. Should all the behaviors broken out by decomposition be retained in the class of the method from which they are decomposed? Should these behaviors be implemented as auxiliary methods? Should designers avail themselves of opportunities (offered by some OOPLs) to include behaviors as stand-alone functions and macros? Should non-behavioral considerations enter into the allocation of these behaviors in a hierarchy (for example, influencing binding time)? In short, structured techniques can be used to improve poorly designed methods, but this does not automatically translate to better designed classes and hierarchies.

Subclass responsibility appears to be a specialized instance of management/work separation. The ancestor class method decides some form of behavior needs to be invoked, descendent class methods actually implement the work. Management should be executed without knowledge about who (or, more accurately, what) performs the work. Designers can therefore employ structured techniques to isolate management/work relationships among functionally cohesive behaviors. Work can then be delegated to descendent classes for implementation.

V. PARADIGM CONFLICTS

The OO philosophy concentrates thought about software development directly on those concepts which most forcefully impact the efficiency of the process - modularization, abstraction, information hiding, reusability, extendibility, and maintainability. It was noted in the previous chapters that the conduct of OOA, OOD, and OOP can vary widely depending upon the notations, methods, and concepts used. OO practitioners must consequently bear much of the burden for producing software that realizes the favorable properties comprehensively. This, in turn, implies that programmers and designers should obtain a sound understanding of potential language mechanism conflicts and design tradeoffs.

This chapter highlights OO language features which can potentially undermine the effective use of information hiding - the principle means by which long-term maintenance costs can be controlled. Specifically, attention is drawn to the encroachments on information hiding produced by inheritance[1]. Additionally, consideration is given to design criteria for employing composition over inheritance.

[1]The information hiding/inheritance conflict reflects design tradeoffs that must be made between information hiding and reusability.

A. ENCAPSULATION VULNERABILITIES

A certain degree of economy enters into the design of OO software. Abstraction, information hiding, and reusability can be viewed as interdependent variables whose values designers collectively attempt to optimize. What constitutes a collective optimum, of course, is reserved to particular design philosophies. Nevertheless, there is a predisposition to consider information hiding as central in any solution to many software lifecycle problems[2].

Abstraction abets thinking that emphasizes essential properties over mundane details. From the outset, abstraction directs attention away from implementati n details. Hence, abstraction supports information hiding in the sense that the abstract conceptual approach promotes design organization which distinguishes property from detail. OO designs and programs exercise their abstract qualities through the respective class interfaces. Consequently, designers must understand OOPL mechanisms and vulnerabilities that circumvent or undermine the strict enforcement of communication controlled by interface.

Inheritance increases code reuse. To the degree that inheritance mechanisms depart from interface enforcement,

---

[2]Software engineering evolved during the late 1970's and early 1980's in large part due to the explosion of software maintenance costs over the product life-cycle. (Booch, 1987)

reuse is achieved at a cost: the internal details of classes are exposed. Even in situations of strict interface enforcement, inheritance creates linkages among related classes that require careful attention when modifications are effected. These reservations with respect to inheritance are particularly prominent for complex applications involving highly developed class hierarchies.

1. External Interface

The external interface consists of those object features available to object users (see Chapter III on external clients). Poorly designed or inadequately enforced external interfaces can lead to a reduction in information hiding.

a. Representation Access

The primary purpose of encapsulation is to hide the structural details of objects. Limiting access to object variables to accessor methods allows designers a finer degree of control: (1) no access, read only access, or read/write access/update methods can be implemented; (2) users need not have any knowledge of variable types; (3) polymorphism can be exploited to construct conversion methods for handling variables[3]; and, (4) variables can be renamed, removed, or

[3]An example would be an object which tracks location. The actual variables may be placed in a Cartesian grid. A polymorphic accessor method can be designed which uses either Cartesian or polar coordinants to set the value of the

reinterpreted[4] without necessitating a recompilation of user code. (Snyder, 1986)

OOPLs or designers may fail to insulate object variables from direct access/update in several ways. First, a language may not offer mechanisms to render the structural details private (i.e., they do not encapsulate). Second, though a language may provide an interface to access/update variables, it may not restrict the user to this interface. OOPLs may allow direct access/update by variable name (such as Simula), or may allow direct access/update through dot notation (such as C++). Third, designers may write methods which contain embedded direct references to variables. In this case, users maintain an indirect capability for representation access. This leads to the curious, if not obvious, idea that objects must be protected from themselves!

b. Creation and Initialization

Some OOPLs provide shortcuts for object creation and initialization which expose implementation details. (Micallef, 1988) Simula, which employs formal parameters, provides initialization by actual specification of values for

---

location variables.

[4]Variable modification/elimination, however, may require reviewing the implementation of the accessor methods to ensure that contracted behavior is maintained. This class method inspection necessity expands if other methods in the class can access variables directly (by name rather than through an accessor method).

the formal parameters; consequently, the "number, type and semantics of formal parameters are a part of the object's external interface." (Micallef, 1988, pg. 18) Flavors allows initialization methods which directly use variable names as keywords. (Micallef, 1988) The preferred course to pursue in these instances is to separate object creation from object initialization such that variable access is limited to the body of initialization methods. (Micallef, 1988) Again, maximum information hiding is achieved when initialization methods must use accessor methods to assign values.

c. *Auxiliary Methods*

As noted in Chapter II/Section B, auxiliary methods are supporting operations, knowledge of which end users do not need. These methods should therefore not be part of the external interface. An OOPL should provide a mechanism to render these methods private to respective instantiations of the object.

2. Internal Interface

The internal interface consists of those ancestor features available to descendent classes by virtue of inheritance mechanisms. Some languages offer the capability of designating class features as private to instantiating clients, but visible to inheriting clients[5]. (Micallef, 1988) As with poorly designed/enforced interfaces to instantiating clients, similarly weak interfaces to inheriting clients can expose implementation details. This subsection considers interface vulnerabilities. Subsequent sections discuss other facets of inheritance which pose problems for information hiding.

a. *Representation Access*

Access to superclass structural details should be limited to accessor/update methods for the same reasons as those outlined in the previous section. Consequently, descendent classes should not be able to directly access/update superclass variables by name or dot notation. This implies that superclass designers must cooperate by including the appropriate access/update methods. (Snyder, 1986)

b. *Embedded Direct Access*

Methods which can potentially be inherited should utilize access/update methods for references to variables embedded in the methods; again, this results in reduced linkages thereby minimizing the effects of variable modification/elimination. Hence, embedded direct access/

[5]Specifically, superclass features declared public or protected are visible to descendent classes. See Chapter II/Section C for more on inheritance.

update is dangerous to both superclasses and descendent classes.

c. *'Self'/'This' Invocation*

Some OOPLs offer devices by which an object can directly invoke methods on itself. Smalltalk employs the word 'self', "...a special variable representing the object which is the receiver of a message." (Smalltalk/V286 Tutorial Programming Handbook, 1988, pg.70) Similarly, C++ uses the word 'this', "a pointer to the object for which a member function is invoked...." (Stroustrup, 1987, pg. 137) Problems arise, however, if operations invoked through this device are redefined by a class or any of its descendants. (Snyder, 1986) An inherited method using 'this' (or 'self') may therefore invoke descendent class methods instead of the intended superclass method.

Several options can address the problem: (1) designers of descendent classes can be aware of inherited method implementations - an undesirable violation of information hiding; (2) superclass designers can limit use of 'self' or 'this' to refer to private methods; and, (3) some other language mechanism can be developed which allows an operation invocation to specify the appropriate superclass. Smalltalk provides a partial solution, allowing the word 'super' to denote operation invocation on a subclass' superclass[6]. Smalltalk/V286 Tutorial Programming Handbook, 1988) C++ uses a scope resolution operator, '::'[7], to specifically designate the source class for method implementation. Use of these devices ('self', 'this', or 'super') unavoidably exposes inheriting classes to modification linkages. Designers should therefore carefully consider whether the convenience of these devices merits potential information hiding lesions.

3. Name Conflicts

Some analysts maintain that name conflicts "...are the root of the inheritance/encapsulation problem as it exists in most OOP languages." (Nelson, Moshell, and Orooji, 1991, pg. 220) Complications occur when descendent classes override ancestor variables. The risk is that overriding may happen

---

[6]This solution appears to push the problem upward one level in the class hierarchy. Suppose a subclass A invokes a superclass B method using 'super'. Superclass B itself employs a 'super' invocation to its superclass C. However, the 'super' call in B to C is to a method which has been redefined in superclass B. The designer of subclass A has no way of knowing (barring examination of implementation code) whether the method he would intend to be invoked is in fact the one selected. This problem arises from the fact that methods using 'super' can be inherited while 'super' only refers to a subclass' immediate superclass. This solution also fails when multiple inheritance is used. (Snyder, 1986)

[7]A superclass X implementation of method Y could therefore be invoked by descendent class Z using the syntax X::Y. Direct naming, of course, links inheriting classes to the named class, exposing inheriting classes to modifications effected on the named class. (Stefik, 1986)

unintentionally - designers may be unaware of ancestor naming conventions because inherited variable names are hidden. Method name conflicts also present potential problems. It appears, therefore, that subclass designers must possess knowledge about the internal details of ancestor classes. This section reviews name conflicts as they apply to single inheritance hierarchies. Multiple inheritance name conflicts are discussed in Subsection 5 of this section.

a. Variable Name Conflicts

As a class hierarchy expands, variable name conflicts can become more involved. In most conventional OOPLs, new variables with the same name as inherited variables are assumed to redefine variables which would otherwise have been inherited. (Nelson, Moshell, and Orooji, 1991) Designers consequently need to be able to distinguish viable redefinitions (overriding) from new variables. Hence, "...the designer of a class must know all that there is to know about the variables inherited from the superclass." (Nelson, Moshell, and Orooji, 1991, pg. 220) Note that this problem also applies to inherited methods with embedded references to variables which have subsequently been redefined.

b. Method Name Conflicts

Method name conflicts foster ambiguities similar to those associated with variable name conflicts. Overriding may unintentionally occur if subclasses define methods using the same names as those of methods private to the parent class. A subtle permutation of this problem is that there may be no way to control methods used by inherited methods. (Nelson, Moshell, Orooji, 1991) An inherited method may contain an embedded invocation to a method which has been redefined. Which implementation is subsequently used is language dependent[8]. (Nelson, Moshell, and Orooji, 1991)

c. *Name Conflict Remedies*

Several strategies can be employed to alleviate or eliminate unintended name conflicts. First, the OOPL environment may include a class hierarchy browser which permits investigation of inherited variable and method names[9].

---

[8]Nelson, Moshell, and Orooji (1991) raise another interesting (and amusing) issue: every variable and method in an ancestor class may be overriden by the time a distant descendent class inherits. Consequently, a class can be an ancestor and yet not supply one inherited feature to the descendent class. Though the resolution of this is a matter of design philosophy, it raises questions as to the nature of the specialization which is being designed into the hierarchy.

[9]This can be considered a violation of information hiding. In particular, it may be the case that descendent classes are designed by teams whose only intercommunication consists of knowledge about the external and internal interfaces. Nevertheless, hierarchy browsers are a common tool which facilitate a potentially simple solution to name conflicts. Information hiding should not be carried to such extremes that development is hindered more than assisted. This thought leads to another, larger issue. Though information hiding, abstraction, and modularization serve to promote reusability, the idea that reuse should be attempted without regard to inherited implementation details ought to be approached with some incredulity. An obvious example concerns code whose failure or aberrant behavior can produce life or system threatening results.

Designers are then free to make informed decisions about overriding. Second, multiple copies of inherited variables with the same name can be maintained, each accessible only by methods inherited through the internal interface[10]. (Nelson, Moshell, and Orooji, 1991) Inherited methods avert name conflicts by continuing to function under the interface environment that existed in the respective superclass. Third, the logic of inheritance can be restricted to extension only. Language facilities can be structured which catch and disallow name conflicts. This solution, though feasible, would overly constrict the inheritance process. Moreover, it would disallow overriding, a mechanism that is sometimes central to specialization guiding the hierarchy construction.

4. Hierarchy/Lattice Modification Problems

Chapter II/Section C reviewed several strategies for shaping class hierarchies. Modifications to a class hierarchy potentially rupture the underlying hierarchy logic, invalidating the contracts which exist between superclass and subclasses. Information hiding cannot completely insulate

[10]Note that the authors have defined the internal interface to consist "...of those methods defined locally for the class and all of the methods in the external interface of each superclass (but not each ancestor) of the class." (Nelson, Moshell, and Orooji, 1991, pg. 223) Name conflicts are avoided by attaching the superclass name to inherited methods. The authors use the term "enheritance" to describe this form of encapsulated inheritance. (Nelson, Moshell, and Orooji, 1991, pg. 223)

classes related through inheritance from the adverse impacts produced by these modifications. Consequently, OO programmers need to understand these linkages before attempting to modify class hierarchies. This caveat applies especially during the maintenance phase of the software life cycle[11].

a. *Inheritance Visibility*

Inheritance visibility refers to "...whether or not the use of inheritance itself should be part of the internal interface (of the class or the objects). In other words, should clients of a class (necessarily) be able to tell whether or not a class is defined using inheritance?" (Snyder, 1986, pp. 40-41) At issue is whether inheritance should remain strictly a mechanism for code reuse, or whether it should enforce particular inheritance strategies (e.g., specialization and/or subtyping). Snyder (1986) and Micallef (1988) contend that inheritance visibility undermines information hiding and reduces programming flexibility.

(1) Excluding operations. "Most object-oriented languages promote inheritance as a technique for specialization and do not permit a class to 'exclude' an

[11]An approach to this problem from a different perspective argues that program-based testing of proven code needs to be reexamined when class hierarchies are modified. (Perry and Kaiser, 1990) Though the discussion is technical, the authors note various linkages produced by inheritance that require retesting of code in both modified and inheriting classes when code is modified.

inherited operation from its own internal interface." (Snyder, 1986, pg. 41) The idea expressed is that specialization ***requires*** that ancestor features be inherited. Hence, a chain of transitive relationships is set up in a hierarchy. A modification such as redefining the superclasses for an ancestor class severely impacts descendent classes built on the expectation of inherited features from the now absent superclasses. Hence, superclass modifications must account for inheritance relationships by maintaining a stable interface.

(2) Subtyping. It was noted in Chapter II/Section C that some OOPLs identify subtyping with inheritance to facilitate static type-checking. "If subtyping rules are based on inheritance, then reimplementing a class such that its position in the inheritance graph is changed can make clients of that class type-incorrect, even if the external interface of the class remains the same." (Micallef, 1988, pg. 25) For example, suppose that class Y is a subclass of class X, and that class Y is redefined to be a subclass of class Z (and not of class X). Objects of class X can no longer be substituted for objects of class Y. Consequently, reusability is reduced and source code may need to be rewritten.

(3) Remedies. Inheritance troubles traceable to subtyping mechanisms (in which types are identified with

classes) cannot be ameliorated[12] - removing ancestor classes will always invalidate existing substitutions based upon type relationships (barring coercion). Clearly, such modifications should occur during the initial design (or rapid prototyping) phase of development, and not during the maintenance phase of mature software products[13].

Two methods are available for reducing inheritance linkages. First, modifications should preserve a stable interface. This leads to the notion that once a class moves into its 'post-production phases', its interface should be closed (except for extensions)[14]. Second, inheritance visibility should be limited to a subclass' immediate superclasses, and inherited methods should only invoke other inherited methods[15]. (Nelson, Moshell, and Orooji, 1991) The

---

[12]It has been noted that subtyping problems could be handled by separating the type hierarchy from the inheritance hierarchy. (Micallef, 1988) Furthermore, "...a formal semantic specification of behavior is needed to be able to correctly do behavioral subtyping." (Micallef, 1988, pg. 27) That is, some standard needs to be agreed upon to relieve programmers of the burden for selecting subtyping rules, thereby automating subtyping decisions.

[13]Note that elimination of any ancestor class from which features are inherited poses the same problem - regardless of the logic guiding hierarchy construction.

[14]It is possible to 'eliminate' methods by coding null implementations. This is a dangerous practice, however, that can produce deleterious results.

[15]Basically, this solution amounts to renaming the offending methods.

latter technique prevents subclass methods from 'reaching up' a hierarchy to ancestor variables or methods which have been overridden by superclasses. In this manner, inheriting classes are concerned only with inherited behavior, and not with inherited implementations. (Nelson, Moshell, and Orooji, 1991)

5. Multiple Inheritance

All of the inheritance problems considered above also apply to multiple inheritance. In distinction to single inheritance, however, solutions are far more complex - individual OOPLs can create inheritance graphs that are unknown and undesired. As an example, CLOS (Keene, 1989) employs specificity rules for determining which specifiers for individual slots will be inherited. Consequently, the definition of a slot may represent an amalgam of specifiers inherited from different classes. Designers have to investigate every superclass to determine the actual form of inheritance. This example returns to the larger problem addressed in Chapter 2/Section C: what precedence rules does an OOPL apply for determining what is inherited and for resolving name conflicts? "The way this conflict is resolved in some languages produces different results if the inheritance graph is changed, even though the external interfaces of the objects remain the same." (Micallef, 1988, pg. 26)

For languages which permit MI, precedence rules determine the shape of inheritance and resolve name conflicts. Generally, such rules either flatten MI graphs into linear chains by introducing a total ordering among the classes and then applying rules for single inheritance[16], or directly locate in the MI graph inheritable variables and methods[17]. (Stefik and Bobrow, 1986)

Linear solutions will produce results which depend upon the decision criteria adopted for arriving at a total ordering. A risk is that these criteria result in chains which do not reflect designer intentions[18]. (Stefik and Bobrow, 1986) Hence, name conflict resolution (and inheritance in general) may not follow designer intentions. This compels the designer to investigate the implementations of actually inherited features to ensure consistency of purpose.

Graph-oriented solutions require the development of graph traversal strategies to implement inheritance. Additional rules must be developed to handle name conflicts

---

[16]Such strategies are titled "linear solutions." (Stefik and Bobrow, 1986, pg. 43) The authors note that Flavors and CommonLoops use this strategy.

[17]Such strategies are titled "graph-oriented solutions." (Stefik and Bobrow, 1986, pg. 42) The authors note that Trellis/Owl and extended Smalltalk use this strategy.

[18]For example, a chain may be established in which an immediate superclass is separated from a subclass by other classes which redefine methods in the superclass.

and inheritance from a common ancestor (non-tree graphs). Name conflict resolution possibilities include the following: (1) do not allow name conflicts; (2) force the subclass to implement variables/methods which override any name conflicts; and, (3) allow inheritance of all conflicting methods[19]. Non-tree inheritance can be handled by limiting inheritance to one set of inherited variables/methods from common ancestors, or multiple sets (depending on the number of paths from an ancestor to descendent class). The problem with graph-oriented solutions is that inheritance is exposed to modifications in the class hierarchy (i.e., inheritance visibility). Furthermore, designers must understand how potential ancestors implement methods (where name conflicts exist) to decide upon which inheritance strategy to pursue.

The solutions to MI problems are the same as those advanced in the previous section: a stable interface and inheritance limited to immediate superclasses. Common ancestor problems are probably best handled by matching the number of instance variable sets to inheritance paths[20]. Name

[19]Such strategies require some rules for deciding how the inherited methods will be invoked. Possibilities include language determined orderings, and designer determined orderings.

[20]This eliminates a situation in which multiply inherited methods from a common ancestor repeatedly update a single set of instance variables also inherited from the same ancestor. (Stefik and Bobrow, 1986)

conflicts can be handled as before - tagging each such method with its superclass name and restricting its visibility to the interface of its class.

6. Design Recommendations

Designers should endeavor to understand the complications introduced by inheritance and minimize its detrimental aspects early in the design phase. Aside from general problems attributable to inheritance, problems specific to individual OO languages should also be thoroughly understood. We now offer the following design recommendations for guiding inheritance decisions:

General Hierarchy Prescriptions -

- Avoid constructing hierarchies which include multiple paths to common ancestors.
- When utilizing multiple inheritance, minimize the number of immediate superclasses.
- If the OOPL being used allows user defined precedence ordering[21] for MI, avoid modifying superclass orderings unless absolutely necessary.
- Reduce the effects of modifications by limiting the depth of hierarchy graphs to three or four levels.
- Always restrict variable access/update to methods in the appropriate interface.

[21]Many OOPLs (such as Smalltalk) determine a precedence order based upon the textual order in which superclasses are listed in the definition of a subclass. Note that this exposes such subclasses to potentially unintended side-effects should the list be reordered.

- Minimize the number of methods in both the external and internal interfaces.

- Exploit public, protected, and private mechanisms, if available in the OOPL being used. If not available, attempt to directly design and enforce these facilities into the class hierarchy.

- Minimize the use of self-referential devices such as 'self', and 'this'.

- Minimize/avoid use of devices which allow direct invocations of variables/methods which are not part of the internal interface (e.g., dot notation).

- Thoroughly understand ancestor class visibility in the OOPL being used. If prominent, carefully consider the consequences of changes to the graph structure.

- Maintain stable class interfaces.

- Minimize the number of embedded method invocations.

Name Conflicts -

- When possible, employ configuration management techniques which limit naming conflicts. If available, exploit class browsers to uncover name conflicts.

- Carefully investigate the resolution mechanisms employed by the OOPL being used. Design classes and modifications with these mechanisms in mind. For many languages, this unavoidably requires implementation visibility of the interface.

- If possible, inherit all variables for which name conflicts exist, using some syntactical device such as tagging with the superclass name to differentiate such variables. Limit interactions with these variables to methods inherited from the corresponding superclass.

- Ultimately, OOPLs will have to be designed which can search inheritance graphs for name conflicts, and make decisions based upon semantic properties of the affected methods and inheriting subclasses.

B. COMPOSITION AND INHERITANCE

A fundamental task of OOD is to determine the behavior and structure of objects (as abstracted into a class). Often, designers must consider whether structures and behaviors should be inherited or realized through composition[22]. This task inevitably requires investigating the nature of the relationships which exist between objects. At first glance, the decision appears to be a simple distinction between 'kind-of' and 'part-of' relationships. However in many situations the complexity of object relationships vitiates quick determination of appropriate relationships, and therefore complicates inheritance/composition choices.

The real difficulty in dealing with composite objects is that the OO design process focuses on classes and class hierarchies in a manner which emphasizes the independence of the abstractions being described. This conceptual approach does not directly lend itself to the analysis of composition relationships which can exist between objects of classes in different hierarchies. Designers must therefore carefully detail the entire nexus of object interconnections that formulate a composite object. A critical matter in this

[22]Recall that inheritance reflects relationships between classes while composition reflects relationships between objects.

regard is determining whether an object's behavior is modified when it forms 'part-of' a higher level object (i.e., has a role to play in some form of collective behavior). Designers must address where these constraints are to be effected, and what policy to pursue if an object has more than one role to play (i.e., is 'part-of' more than one object).

This section reviews composition, advances criteria for recognizing composition relationships, draws out the implications of using composition versus inheritance, illustrates inheritance/composition tradeoffs by presenting solutions to a simple design problem using three different OOPLs, and considers to what degree composite objects should parallel the details of real-world objects.

1. Composition Reviewed

Chapter II/Section D introduced the idea of composition, noting several properties of composition relationships. The concept is further analyzed in this subsection; particular attention is directed to subtleties which complicate the design of composite objects.

Composition "...is a tightly coupled form of association[23] with some extra semantics." (Rumbaugh, 1991, pg. 37) Rumbaugh (1991) discusses several properties of composition which distinguish it as a specialized form of

---

[23]Association is an abstraction used to group objects from several independent classes. (Elmasri and Navathe, 1989)

association: (1) the relationship is transitive[24]; (2) the relationship is antisymmetric[25]; and, (3) properties (state values and operations) of the whole can propagate to the part[26].

That properties propagate raises several interesting issues. First, the form of interconnectedness induced by composition requires analyzing the existential dependency of contained objects. Specifically, should an object which is part of another object manifest a separate identity? This has implications for creation and destruction operations. It also portends the possibility of conflicting interobject interactions. Technically, the problem can be viewed as one in which the variables which make up an object are themselves objects[27], or are pointers to objects[28]. (Nelson, 1990) Not surprisingly, pointer referencing also allows contained objects to be shared by more than one containing object. Second, subobjects and dependent objects most likely will have

---

[24]For example, A is part of B, and B is part of C implies that A is part of C.

[25]A is part of B implies that B is not part of A.

[26]An example would be that the speed of an aircraft propagates to the parts which compose the aircraft - the wheels, engines, wings, etc.

[27]Nelson terms these "dependent objects." (Nelson, 1990, pg. 5) Dependent objects do not exist apart from the objects they are a part of.

[28]Nelson terms these "subobjects." (Nelson, 1990, pg. 5) Subobjects do exist apart from the objects they are part of.

some properties in common with the object of which they are a part[29]. This follows naturally from the idea of property propagation. However, as suggested above, subobjects may be 'part-of' more than one object - creating a requirement to insulate subobjects against potentially conflicting state changes. Third, transitivity implies that these considerations flow through to all objects at whatever level in a composition hierarchy[30]. Complex determinations must be made as to what level properties propagate to. Finally, antisymmetry implies that properties do not necessarily propagate to higher levels in a composition hierarchy.

Another facet of composition requiring attention is the cardinality relationship between containing objects and sub/dependent objects. Possibilities include the following:

> *Aggregation can be fixed, variable, or recursive. A fixed aggregate has a fixed structure; the number and types of subparts are predefined[31]. A variable aggregate has a finite number of levels, but the number of parts may vary[32]. A*

---

[29]This possibility reflects a design consideration - should propagated properties be maintained by contained objects, or can these properties be assumed to be maintained by the containing object? Solutions to this question may be influenced by whether or not contained objects exist independent of the containing object.

[30]A composition hierarchy can be conceived of as a tree depicting composition relationships. Such a hierarchy starts with the highest level object, and moves downward to successively more granular levels of detail.

[31]For example, a golf club always has one clubhead, one shaft, and one grip.

[32]For example, an academic course may consist of a single professor and several students. However, there may be a variable number of students per course (i.e., a one-to-many

recursive aggregate contains, directly or indirectly, an instance of the same kind of aggregate; the number of potential levels is unlimited[33]. (Rumbaugh, 1991, pg. 59)

Fixed aggregates are the easiest to understand and design. Variable aggregates, however, pose interesting problems for reusability. How does one design a class for composite objects such that objects can be instantiated with variable numbers of parts? Must each variation be modeled by a distinct class (using inheritance)? This problem will be investigated in Subsection 4.

Finally, a distinction is sometimes drawn between objects that are composed of other objects (a car) and objects that contain other objects (an array of integers). (Wirfs-Brock, Wilkerson, and Wiener, 1990) Though both relationships can be modeled as composition, containment is a much looser form of association. It is often the case that a container does not need to interact with the elements it holds. In other cases, interactions do occur[34]. Frequently, containment relationships involve a design choice between employing composite objects or instantiations of parameterized classes (see Chapter

---

relationship prevails).

[33]For example, a computer program may consist of blocks containing compound statements which, in turn, contain other blocks. (Rumbaugh, 1991) Recursive composition should be avoided in most cases as there is the potential for an infinite recursion in which an object calls upon itself to formulate its definition.

[34]For example, a hash table may need to ask an element for its hash code before adding the element to the table. (Wirfs-Brock, Wilkerson, and Wiener, 1990)

II/Section B). Containers that do not interact with their parts are probably best modeled using parameterized classes. This clearly identifies the limited behavioral connections between the container and the objects it holds.

2. Recognizing Composition

Several keys to composition have already been discussed: (1) 'part-of' relationships; (2) propagation of properties/operations; (3) cardinality considerations; and, (4) interobject behavioral constraints[35]. Other facts of a problem situation may also suggest that composition fits a particular object to object relationship. These include the following: (1) collective instantiation and destruction; (2) delegation of responsibilities from the whole to parts; (Wirfs-Brock, Wilkerson, and Wiener, 1990) and, (3) service unity[36].

3. Composition or Inheritance?

Though individual facts may indicate that composition is the appropriate design choice, the possibility always exists for modeling such relationships using inheritance[37]. In

---

[35]Composite objects "...describe for instantiation a richly connected set of objects...." (Stefik and Bobrow, 1986, pg. 58)

[36]This rather vague notion marks the fact that parts do not act independently, but rather are controlled by the unifying purpose of the whole object. This would serve to distinguish, for instance, a clock and a radio which happen to be collocated, from a clock-radio.

[37]In particular, multiple inheritance can be used as an alternative to composition.

many situations, it is not clear whether composition or inheritance should be used (the design problem considered in the following subsection is an example). Hence, it is worth considering in what respects the consequences of selecting one over the other differ.

Inheritance is a class relationship, whereas composition is an instance association; differences in the nature and mechanics of the two relationships start from this fundamental distinction[38]. First, the hierarchical structure which serves to define a class through inheritance establishes the identity of a single object. Composition, on the other hand, involves a relationship between objects with separate identities[39].

Second, composition strictly limits visibility on the part of the containing object to the external interfaces of its parts. Inheritance allows a finer degree of visibility which generally entails greater accessibility through the internal interface.

Third, inherited behavior is visible to other objects to the extent that it is included in the external interface. The behavior of objects serving as parts, on the other hand, is not visible to other objects - the containing object

---

[38]Composition can be thought of as a form of part inheritance while inheritance can be viewed as behavioral inheritance. (Nelson, 1990)

[39]This holds regardless of whether the part is a dependent object or a subobject.

mediates any such interactions[40].

Fourth, "behavior can be easier to reuse as a component than by inheriting it." (Johnson and Foote, 1991, pg. 124) As an example, it is easier to add an extra scrollbar to a window as a component, than it is to multiply inherit scrollbars. (Johnson and Foote, 1991) The idea is that inheritance used in this manner may require undesirable changes to the behavior established by ancestor classes.

Fifth, as noted in the previous section, inheritance potentially exposes information hiding to compromise. Composition does increase the coupling between objects; nevertheless, information hiding is not violated since all interactions are managed through the respective external interfaces.

Finally, specialization was previously identified as the principle strategy for structuring class hierarchies (see Chapter II/Section C). Hence, an appropriate question to ask when contemplating inheritance/composition choices is whether or not a resulting subclass can be said to be a specialization of the class(es) from which it inherits. For example, "it is not valid to define a class Car that inherits from Body, Frame, Wheels, and similar classes, since a car is not a wheel." (Nierstrasz, 1989, pg. 8) A different perspective on specialization is to consider the substitution possibilities

[40]This distinction is reduced to the extent that an OOPL permits manipulations by pointer operations.

discussed in Chapter II. Inheritance should not be used if substituting a subclass object for a superclass object does not satisfy applicable compatibility requirements (for example, a car cannot be substituted for a wheel in any meaningful way).

4. The Clock_Radio Problem

It is instructive to underline the ramifications of selecting composition or inheritance by investigating the solution to a sample design problem. This section considers designs for a clock_radio in three different OOPLs: C++, Smalltalk, and CLOS. Separate solutions for each OOPL, one using inheritance and the other using composition, are illustrated and compared.

Although a major theme of this thesis is the advocacy of OOA/OOD practices which enhance the entire OOP process, independent of any particular OOPL, it must be conceded that at this point in the evolution of OOPLs language selection does impact design opportunities. Hence, the three OOPLs are also used to illustrate language dependent differences between composition and inheritance.

a. *Problem Statement*

The clock_radio is a common household device which consists of a clock and a radio. It manifests many of the properties listed above for recognizing composition: (1) the clock and the radio are parts of a clock_radio; (2) properties such as location and power propagate from the whole to the parts; (3) a clock-radio could conceivably contain more than one clock, or more than one radio; and, (4) behavioral constraints not ordinarily associated with separate clocks and radios are possible (for example, a clock controlled timer can turn the radio off). Figure 2 displays the responsibilities that rudimentary OOA might reveal as germane to a clock_radio. State information represents knowledge that a clock_radio maintains about itself. Services define the behavior of a clock_radio. The list is obviously not complete, but for present purposes can be accepted as a standard upon which permutations may be structured. Note that Figure 2 is not a class definition in any OOPL, but rather a listing of behaviors and state information. Hence, distinctions such as instance and class variables are not required.

```
State information
   present_location      radio_alarm_time
   power_on              clock_alarm_time

Services
   set_time              volume_increase
   set_clock_alarm       volume_decrease
   set_radio_alarm       select_channel
   play_am               radio_on
   play_fm
```

Figure 2: Clock_Radio Responsibilities

In the subsections which follow analysis will focus on three classes: electric_clock, radio, and clock_radio. To simplify discussion, the electric_clock and radio classes are not placed into hierarchies (i.e., a detailed design in which suitable variables and methods are elevated to abstract classes is not performed). Figure 3 illustrates the inheritance class relationships, and Figure 4 shows the composition object relationships which will be modeled.

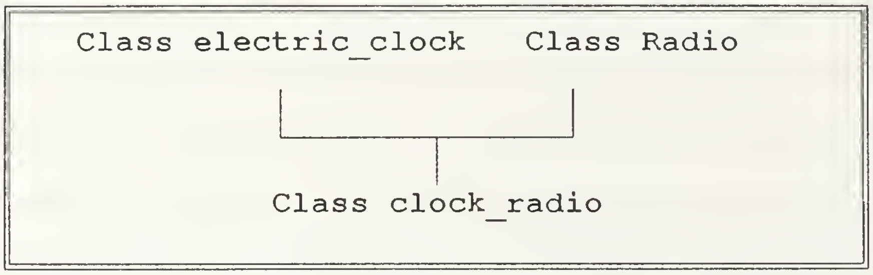

Figure 3: Inheritance Class Relationships

```
Object clock_radio

          object electric_clock
          object radio
```

Figure 4: Composition Object Relationships

The implementations contained in Appendices A, B, and C are not intended to demonstrate a comprehensive, usable solution. In particular, error handling is not provided. For the dynamically typed languages (Smalltalk and CLOS), no effort was made to enforce type checking. Casual observation will also reveal that instance variables are directly accessed in most of the implementations. This was done to reduce and simplify the code.

b. C++ Solution

C++ is a hybrid OOPL, erected on the foundation of the C programming language. The principal building blocks include objects, message passing, classes, and inheritance hierarchies. C++ is a strongly typed language in which classes implement abstract data types. The language does provide for virtual classes and for public, protected, private visibility declarations. Multiple inheritance hierarchies can also be built.

Appendix A contains the C++ solution to the clock_radio problem. The code successfully compiled and tested on a Borland Turbo C++ compiler. (Borland International, 1990) Section A contains the declarations for the various classes in the problem, including classes for types time and position. Following the class name is a list of type declarations (for example, x_y_posit) and variable names (for example, radio_location). Due to the absence of any visibility declaration, the default visibility for the variables is private. Note that variables such as size and color could also be included. These were omitted since location and power already serve the purpose of demonstrating variables whose value propagate from the whole.

All methods are given public visibility; consequently, they belong to both the external and the internal interfaces. 'Void' indicates that a method does not return a value. Finally, a method with the same name as the class name is a constructor. Constructors set aside space in memory when an object instance is created and can be used for initializing instance variables[41]. Similarly, a method with the class name and a tilde (~) prefixed is a destructor. An example of a destructor is given in electric_clock class. Destructors, which can only be called by the compiler, are used to undo side effects such as changes to global variables. (Eckel, 1989) Generally, programmer-defined destructors are not included; instead, a default destructor supplied by the compiler is used. (Eckel, 1989)

(1) Inheritance. Class clock_radio in Appendix A/Section A illustrates the design of a clock_radio class using multiple inheritance. The public declarations in the first line indicate that the internal and external interfaces of superclasses are inherited as designed (variable/method visibilities remain unchanged). Sections B through G provide implementations of the methods for the respective classes. Observe that a more fully developed clock design would access operating system clock functions to provide actual time behavior. This was not done to simplify the problem. Note the efficiency by which the behavior of the class is managed through multiple inheritance. Simple extension is used to complete the behavior of the class, and reusability is exploited.

---

[41]In this case, it is assumed that the constructor initializes the electric-clock to a powered state and the alarm to off (i.e., power_on := True, and alarm_on := False).

Several other features merit attention. First, clock_radio is a subtype of both an electric_clock and a radio, and can consequently substitute for either one. This follows from the identification of class and type hierarchies enforced by C++.

Second, although not present in this design, it is conceivable that name conflicts could exist for the location variable and the power method. C++ resolves this through resolution operators. Name conflicts must always be considered when using inheritance. Unless the design strategies noted in the previous section are followed, preventing and resolving name conflicts inevitably requires knowledge about superclass details.

Third, while C++ allows a class to be inherited indirectly more than once, a given superclass can only be directly inherited once. This poses cardinality problems. Essentially, a clock_radio class based on inheritance must be redesigned each time a different combination of clocks or radios is desired.

Finally, constructor and destructor methods cannot be inherited. (Atkinson and Atkinson, 1991) Unless compiler supplied default constructors and destructors are preferred, subclass methods must explicitly account for ancestor constructors and destructors. This fact assumes importance during hierarchy design as it can impact object initialization and free memory management[42]. (Atkinson and Atkinson, 1991)

(2) Composition. The composite_clock_radio class in Appendix A/Section A uses composition to reuse electric_clock and radio behavior. Several features alluded to above are immediately apparent. First, the external interface of the clock_radio is the only conduit to contained objects for objects using a clock_radio. Hence, greater design effort is required to engineer the desired behavior. Note that the external interface to the composite_clock_radio class essentially duplicates the electric_clock and radio interfaces. Although this promotes information hiding, it also translates to a degree of inefficient CPU use since it amounts to providing methods whose sole purpose is to function as a protective layer (i.e., doubles function call processing).

Second, name conflicts are not a problem. If all interactions are forced through the external interfaces of the electric_clock and the radio respectively, name conflicts cannot occur.

Third, the facility with which extra radios or electric_clocks could be added as constituent parts is evident: simply declare new variables of the required types. Nonetheless, the implementation of clock_radio methods which control interactions among the various parts would have to be

[42]It has been recommended that virtual constructors and destructors be declared in superclasses to ease the design burden on subclasses. (Atkinson and Atkinson, 1991)

modified to account for the new structure of the entire object. The same logical changes have to be made when using inheritance. In this respect there is no advantage to using composition. However, it still remains much easier to create multiple parts using composition.

Fourth, as is the case with inheritance, constructors and destructors must be specifically accounted for. This can be done comparatively easily using initialization lists. (Atkinson and Atkinson, 1991) In distinction to inheritance, deciding which constructors/destructors to include is simple: those of contained objects (this assumes no inheritance). Multiple inheritance, on the other hand, can produce complicated scenarios in which sequencing of construction/destruction is important. Composition only requires that contained objects be created prior to the containing object.

Fifth, a strategy needs to be adopted for handling properties which propagate. Both electric_clock and radio objects contain instance variables for location. Designers must decide whether or not to include a location instance variable for the clock_radio, and whether or not to propagate location assignments to both the electric_clock and the radio parts. Property propagation creates familiar data update problems (ensuring consistent information is maintained), and constitutes unnecessary data duplication[43]. Notice also that property propagation also implies some knowledge about the details of part objects[44]. In this solution, responsibility for maintaining location and power information is retained by the clock_radio. Changes are propagated to its respective parts. Note, however, that this responsibility is directly inherited when inheritance is used. Consequently, a location instance variable was not created for clock_radios formed by inheritance. This appears to be a situation in which it would be useful to design electric_clock or radio subclasses which exclude the location instance variable. However, all the methods in the affected class would hav to be searched to eliminate direct references to this variable or calls to accessor methods.

Finally, a clock_radio is no longer a subtype for an electric_clock or a radio. Hence, substituting a clock_radio for either of its two parts would generate an

[43]A significant difference should be noted between inheritance and composition. In languages which do not provide visibility control mechanisms like those found in C++, the external interface includes inherited methods. Hence, messages can be sent to descendent objects that invoke inherited methods which return state information relating to propagated properties from instance variables declared in ancestor classes. Designers must ensure that all such state information is current and consistent since such calls can not be precluded in these kinds of languages. Smalltalk is an example of one such OOPL.

[44]All of these considerations also apply to multiple inheritance in which more than one superclass maintains the same state information.

error message at compile time

c. *Smalltalk Solution*

Smalltalk[45] is a dynamically typed OOPL featuring objects, classes, metaclasses, and single inheritance[46]. Messages can be sent to instances (instance methods), or to classes (class methods)[47]. Class definitions include instance variables and class variables. A singular quality of Smalltalk is its comprehensive environment: programming and design are all accomplished within the confines of the Smalltalk system of disk and hierarchy browsers employing extensive windowing and menu controls.

Another useful feature of the Smalltalk environment is its library of predefined classes. All classes (predefined and user defined) comprise one large hierarchy descended from the root class ***Object***. New classes are defined by filling in the appropriate information in system supplied templates. New classes must be descended from a superclass in the hierarchy.

---

[45]Several variants of the Smalltalk language are commercially available. These dialects manifest widely varying capabilities. The present discussion draws upon Digitalk's Smalltalk/V286. (Smalltalk/V286, 1988)

[46]Other versions of Smalltalk do provide for multiple inheritance. (Stefik and Bobrow, 1986) However, the use of multiple inheritance in these versions "...is not used much or institutionalized." (Stefik and Bobrow, 1986, pg. 49)

[47]Classes are treated as objects in Smalltalk; hence, the need for metaclasses.

(1) Inheritance. Though the requirement to place new classes into the inheritance hierarchy facilitates shaping a common protocol, the absence of stand-alone classes reduces the flexibility with which designers can construct classes - it slows development to the extent that designers must consider potentially lengthy inheritance chains. Visibility of variables to other objects is limited to accessor methods provided by the class. All variables and methods are inherited; subclasses can add new variables or methods, and can override superclass methods.

Appendix B/Section A displays completed templates for the various classes in the clock_radio problem. Sections B through G display method implemtations. The code successfully interpreted and tested on Digitalk's Smalltalk/V286 interpreter. The electric_clock class inherits from the radio class, while the radio class is assumed to have the electric_device class (not depicted) as its superclass. Variables are defined without type declarations and the class interface is not included in the template. Methods are defined by selecting the ***new method*** menu option; hence, a method interface and its implementation are defined simultaneously. Each class has the same methods as those previously identified for the corresponding C++ class.

Given that Smalltalk/V286 allows only single inheritance, one of the two superclasses must be inherited from the other in order to replace multiple inheritance. This is not a desirable state because instances of the inheriting class must exhibit behavior of the superclass; hence, for example, a pure radio could not be instantiated should the radio class inherit from the electric_clock class. Additionally, such inheritance establishes hierarchies that do not truly reflect any sort of specialization: an electric_clock is not a specialization of a radio. Designers should be very deliberate and consistent in selecting criteria for structuring the single Smalltalk hierarchy. Problems arise when the same class serves multiple roles: template for inheritance, part object template, and user object template[48].

Immediately, all name conflict problems previously discussed reappear. Note that Smalltalk treats conflicting instance variable names as an error. Note also that a clock_radio can be safely substituted for an electric_clock or a radio using this form of inheritance. Similarly, an electric_clock can be safely substituted for a radio since it inherits behavior from the radio class. The latter possibility is not desirable for reasons presented earlier.

Inherited instance variables are accessible by name in Smalltalk. This poses a problem for designers of inherited classes such as clock_radio. A clock_radio

[48]A user object (my term) is a semantic notion describing objects which interact with other objects through respective external interfaces while maintaining independent identities that do not involve composition. The idea is drawn from Booch who distinguishes containing relationships from using relationships. (Booch, 1991)

designed through inheritance is conceived of as a single, unified object. Given that Smalltalk is a dynamically typed language, a designer must have knowledge of superclass method implementations in order to avoid inconsistent variable typing (especially if overriding inherited variables or methods). Conflicts can be avoided by using only inherited methods to access/modify inherited variables. However, this then requires that ancestor classes be designed to account for descendent class (actual and potential) requirements[49]. This sort of design thinking is adequate for specialization hierarchies, but is exceedingly difficult for hierarchies replacing multiple inheritance.

As is the case with C++, inheritance in Smalltalk is entirely inadequate for designing clock_radios with multiple clocks or radios. The limitation is made apparent by the fact that only single inheritance is allowed.

Finally, designers should design constructor and initializer class methods which account for ancestor classes. The safe approach is to design initializer methods which issue ***super*** calls to the superclass initializer; thereby, avoiding any typing errors. Smalltalk automatically supports garbage collection; hence, destructors are not required.

(2) Composition. Appendix B/Section A displays the Smalltalk definition for a composite_clock_radio class:

---

[49]That is, an ancestor class must provide descendent classes with all the methods they require to access/modify inherited variables.

instance variables for a clock and a radio are defined. All interactions with the objects these variables point to are managed through the objects' interfaces. Clearly, this approach to designing a clock_radio is much cleaner than a solution based upon ersatz multiple inheritance.

Again, the template in Section A illustrates that only variable names have been defined. It has been argued that the absence of a type-system in Smalltalk renders it impossible for a compiler to optimize Smalltalk code. (Johnson, 1988) The thought can be taken a step further by asserting that the absence of a type-system also severely undermines the type continuity that should prevail among the parts of an object. Designers must assume responsibility for ensuring that conceptually inappropriate type assignments or method selections do not occur at run-time. This is, of course, a difficult task at best. However, in the absence of a type-system that can be used by a compiler[50], designers should enforce type continuity of parts through the following steps: (1) clock_radio object creation should create and initialize parts according to their appropriate class; (2) state changes stored in instance variables should be effected

[50]Johnson describes an effort at introducing a type-system to Smalltalk that is "...type-safe, handles polymorphic procedures and parameterized types, and can be used by an optimizing compiler." (Johnson, 1988, pg. 317)

by accessor methods which perform type checking[51]; and, (3) messages sent to a part are type correct for the part, or for any ancestors of a part[52].

Multiple clocks or radios can be added by defining specialized subclasses of class clock_radio. Clock_radio class methods which organize part behavior will need to be extended to accommodate new structures.

d. CLOS Solution

CLOS is a hybrid OOPL built upon the Common Lisp programming language. (Koschmann, 1990) It features objects, classes, generic functions, and methods. Classes consist of local and shared slots (instance and class variables). Programmers attach methods to generic functions through method definition[53], and users invoke methods by calling the

[51]Most Smalltalk instance variables contain pointers referring to objects. (Smalltalk/V286, 1988) Some instance variables contain 8 bit bytes representing elementary data values. (Smalltalk/V286, 1988) Instance variable types can easily be changed by assigning pointers to different objects (i.e., avoid aliasing).

[52]Johnson states that "a message-send is type-correct if it is type-correct for each possible object type of the receiver." (Johnson, 1988, pg. 318) He describes the process of unification which establishes that a "procedure call is type-correct if there is some assignment of types to type variables (of the method) that makes the types of the arguments be in the types of the parameters; the return type of the procedure call is the return type of the definition of the method with all the type variables replaced by the assignment to them." (Johnson, 1988, pg. 318)

[53]Generic functions constitute the interface to a set of methods. Each method has the same name as the generic function it implements, and the same number of parameters. The generic dispatcher selects appropriate methods on the basis of type correspondence between message arguments and method

appropriate generic function. (Keene, 1989) CLOS provides for multiple inheritance in which both slots and methods are inherited. The order in which superclasses are listed in a class definition determines precedence for handling problems such as name conflicts.

CLOS is a dynamically typed OOPL. (Koschmann, 1990) Slots can be typed using the ***:type*** specifier. However, many commercially available CLOS interpreters do not enforce such slot typing. (Keene, 1989)

Appendix C/Section A illustrates the definitions for the various classes in the clock_radio problem. Most of the slots in the clock_radio problem are assigned default values using the ***:initform*** specifier. This prevents any slots from being unbound. Sections B through G display the various method implementations. The code successfully interpreted and tested using an Allegro CL interpreter (Sun4 version).

(1) Inheritance. As is the case with C++, the use of multiple inheritance in CLOS to build a clock_radio class is extremely efficient. A clock_radio class can be descended from these two superclasses as illustrated by the clock_radio class in Section A. Again, the order in which superclasses are listed determines the precedence which the generic

---

parameters. In the absence of a corresponding generic function for a method definition (defmethod), CLOS automatically creates the generic function from the defmethod. The CLOS implementation to the clock_radio problem assumes this automatic generic function building; hence, only defmethods are illustrated.

dispatcher will use in selecting methods for generic function calls. Consequently, this form of precedence exposes implementations to the modification errors described in Chapter V/Section A. All of the reservations expressed with respect to a solution using inheritance in C++ also apply to a solution in CLOS[54]. Additionally, dynamic typing presents the same problem as that noted for Smalltalk - designers need to carefully monitor what information is maintained by slots[55].

(2) Composition. The composite_clock_radio class demonstates composition in CLOS. Essentially, a pointer to the desired part_of object is created (using the built-in function ***make-instance***) and assigned to the slot variable. As in Smalltalk, it is required that slots be created and initialized to the appropriate types. Observe that the parts have been given names (for example, clock_one) in anticipation of instances which have multiple parts. This is not good programming style in that users must maintain information about the parts of an object, and can directly refer to these parts. Although direct naming is not required for a

---

[54]CLOS contains a variety of mechanisms for defining and organizing behavior: mixins, multi-methods, before methods, after methods, around methods, and individual methods. Developers can exercise tight control over method structure, and generic dispatching. However, these opportunities often result in designs which are highly individualized, tightly coupled, and susceptible to modification errors.

[55]CLOS allows slots to be directly accessed using the setf function. Interfaces which include this function directly expose data structures and expose slots to dynamic type changes. In short, designers must guarantee that slots are in fact encapsulated.

clock_radio consisting of only one clock and one radio, such naming appears to be unavoidable for this kind of object when larger numbers of clocks or radios are included. Clock_radio methods can be defined which organize the interactions between clock_one and radio_one. As before, cardinality can be handled by defining composite_clock_radio subclasses which add new parts and override relevant controller methods when necessary.

An important difference from Smalltalk is that slots can be directly accessed/modified using the built-in functions ***slot-value*** and ***setf***. CLOS provides a slot specifier ***:accessor*** which automatically creates a generic function for reading and writing a slot. The accessor so created forms part of the external interface. Nevertheless, slots are not encapsulated by accessors. CLOS does not provide any mechanisms for enforcing the notion of a private slot. Consequently, composite objects in CLOS do not truly form an intermediate layer between users and parts.

5. Composition Granularity

Another interesting problem with respect to composition is the granularity of detail. An example may serve to demonstrate the problem. Each clock class can be designed such that it includes one electric plug/cord combination. Should a clock_radio consisting of multiple clocks therefore have multiple plugs and cords? The problem arises from the fact that classes can serve both as templates for composition objects and as templates for independent objects.

The solution to this problem lies in reconsidering the fundamental nature and purposes of OOA and OOD. Guiding OO development is the goal of specifying and organizing the behavioral properties of objects in the application domain. Although aspects of physical structure enter into designs, these only occur as required to fulfill behavioral requirements. Similarly, variables are included only as required to preserve state information. The primary focus, it should be reemphasized, is on behavior. Hence, a clock is either powered or it is not. This is state information which should be included in a design. Though a plug/cord form a conduit for powering a clock, they really do not have any behavioral responsibilities that other objects may interact with, and they cannot be associated with any state preservation duties. Consequently, a design for a clock class should not include variables for plugs and cords.

The clock_radio problem demonstrated, however, that different classes may maintain identical state information. Hence, multiple instances of the same piece of information are maintained when either inheritance or composition is used. As suggested in the clock_radio discussion, this is an unavoidable problem, excepting exclusion, which requires some sort of data consistency/duplication policy on the part of designers.

The same problem reappears in another context. It has previously been noted that both in the context of system design and in composition design properties/responsibilities are delegated to constituent elements. It is not improbable that delegated properties/responsibilities may not be needed in new design problems; thus, reducing reusability opportunities and forcing considerable redesign. Expanding upon the notion of moving common behavior upwards in class hierarchies (see Chapter III), classes at the higher levels in a hierarchy should provide common behaviors in the most general sense: behavior generalized to the class of objects as they might occur in any application. Hence, a clock class would provide behavior expected of any clock. Subclasses can then be used to extend behavior to suit particular system or composite object requirements. What forms a generalized formulation of behavior applicable to a class of objects is a knowledge problem that must be arrived at during OOA.

VI. CONCLUSIONS

A. SUGGESTIONS FOR FUTURE RESEARCH

1. Specialization

Chapter II introduced the notion of specialization. This concept figures importantly in controlling the development of inheritance hierarchies. Nonetheless, a formalized definition of specialization needs to be researched. Such a definition should distinguish specialization from subtyping, and should explore the interrelationships between the two concepts when building inheritance hierarchies.

2. Reusability

The various design strategies discussed in Chapter III advanced criteria for recognizing and organizing the relationships among the elements of a problem. These strategies provided criteria for the explicit organization of behavior in designed classes. However, the immediate focus of these approaches generally fell upon the current problem. Research needs to be directed toward uncovering criteria for organizing knowledge in a manner that facilitates reusability for potential applications. What level of generality should be placed into abstract classes? At what level in a hierarchy should abstract classes give way to concrete classes? Should consideration of potential subclasses influence the design of concrete classes? What criteria should determine which behaviors define a class of objects, and how do these criteria influence reusability?

3. **Knowledge Allocation**

It was suggested in Chapter V that structured techniques can be used to improve the implementation of OO methods. However, recommendations for accomplishing this were scrupulously avoided. Research needs to be directed at uncovering criteria for using the results of such techniques to improve OO design. In particular, it may prove fruitful to investigate whether criteria other than behavioral correspondence (behavior to real-world object) should influence the allocation of behaviors in a hierarchy.

4. **Functional Decomposition**

Chapter IV analyzed the design practice of subclass responsibility as a subset of techniques descended from functional decomposition. Heuristics were suggested for designing subclass responsible behaviors. Further investigation should be conducted into the potential vulnerabilities of hierarchy designs which use functional decomposition techniques to uncover and organize subclass responsible behaviors. In particular, the ways in which hierarchy modifications can expose such designs to unexpected errors merits research.

B. CONCLUSIONS

The OO paradigm represents a new perspective on the practice of developing software. It arrives with a complement of concepts, tools, and theories which permits developers to organize their thinking in a fashion which parallels the manner in which humans develop and compose knowledge about the real world. As experience with the OO approach progresses, the realization grows that the greatest gains occur in the areas of analysis and design: intelligent, coherent application of OO concepts during these phases of development substantially reduces the effort and costs associated with programming and maintenance.

Though the OO paradigm promises much, the absence of conceptual continuity and standardization has so far resulted in a multiplicity of distinctly diverse OOPLs and development strategies. This thesis has attempted to formulate an understanding of fundamental OO concepts (Chapter II) and the tradeoffs involved in applying those concepts (Chapter IV). Such a foundation should serve to facilitate analysis/design practices (Chapter III and Chapter IV) which realize the many benefits attributable to OO development (Chapter I).

Particular attention has been drawn to inheritance as a vehicle for organizing knowledge. Inheritance smartly applied is a powerful mechanism for reusability. It is the position of the author that among the strategies which can be employed to construct inheritance hierarchies, the safest and conceptually most appropriate course lies in identifying class hierarchies with type hierarchies. This allows the compiler to perform optimization operations, and forces some form of semantic consistency/structure on the inheritance hierarchy. In this setting, developers also need to converge on a practicable definition of specialization.

While inheritance promotes programming economy through reusability, the single most important aspect to OOP (or any programming methodology) is information hiding. Information hiding shields designers from their own mistakes while concomitantly reducing the effort expended in uncovering and correcting mistakes. Designers utilizing inheritance must understand and account for inheritance/information hiding tradeoffs. Negligence in this regard can only lead to long-run maintenance difficulties and attendant cost increases.

APPENDIX A. C++ CODE

A. C++ CLASS DEFINITIONS

```
#ifndef CLKRAD_HPP
#define CLKRAD_HPP

const am    = 0; //define switch values
const fm    = 1;
const true  = 1;
const false = 0;
const off   = 0;
const on    = 1;

class x_y_posit {
   int x;      //Cartesian reference system
   int y;
public:
   x_y_posit (int x_pos = 0, int y_pos = 0) {x = x_pos;
      y = y_pos;}
   x_y_posit operator=(x_y_posit);
   void display_position();  };

class time {
   int hours;      //military time
   int minutes;
   int seconds;
public:
   time (){hours = 0; minutes = 0; seconds = 0;}
   time (int hrs, int mins, int secs){hours   = hrs;
                                  minutes = mins;
                                  seconds = secs;}
   void change_time (time);
   time operator=(time);
   void display_time(); };


class radio {
   x_y_posit radio_location;  //can be implemented many
   int       radio_powered;   //true equals powered
   int       radio_on;
   float     fm_station;
   float     am_station;
   int       am_fm_switch;
   float     volume_level;
public:
   radio (x_y_posit);
   void change_radio_position (x_y_posit);
   void power_radio();
   void remove_radio_power();
   void turn_radio_on ();
   void turn_radio_off();
   void increase_volume ();
   void decrease_volume ();
   void select_am();
   void select_fm();
   void select_am_station (float);
   void select_fm_station (float);
   void display_radio_on_off_state ();
   void display_volume_level ();
   void display_am_channel ();
   void display_fm_channel ();
   void display_radio_position ();
};


class electric_clock {
   x_y_posit clock_location;
   int       clock_powered;
   int       clock_alarm;
   time      current_time;
   time      clock_alarm_time;
public:
   electric_clock (x_y_posit);
   void change_clock_position (x_y_posit);
   void power_clock ();
   void remove_clock_power();
   void reset_clock_time(time);
   void clock_alarm_on ();
   void clock_alarm_off ();
   void set_clock_alarm_time (time);
   void display_clock_alarm_time ();
   void display_clock_time ();
   void display_clock_position ();
   ~electric_clock ();
};
```

```
class clock_radio : public radio, public electric_clock {
    int    clock_radio_powered;
    time   radio_alarm_time;
    int    radio_alarm;
public:
    clock_radio(x_y_posit);
    void change_clk_rad_position(x_y_posit);
    void power_clock_radio();
    void remove_clock_radio_power();
    void set_radio_alarm_time(time t);
    void radio_alarm_on();
    void radio_alarm_off();
    void display_radio_alarm_on_off_state ();
    void display_radio_alarm_time ();
    void display_clock_radio_position ();
 };


 class composite_clock_radio {
     electric_clock clock_one;     //'part_of"
     radio          radio_one;     //"part_of"
     x_y_posit      comp_clk_rad_position;
     int            comp_clk_rad_powered;
     int            comp_radio_alarm;
     time           comp_radio_alarm_time;
 public:
     composite_clock_radio (x_y_posit);
     void power_comp_clock_radio();
     void remove_comp_clock_radio_power();
     void change_comp_clk_rad_position (x_y_posit);
     void turn_comp_radio_on ();
     void turn_comp_radio_off ();
     void increase_comp_radio_volume ();
     void decrease_comp_radio_volume ();
     void select_am_station (float);
     void select_fm_station (float);
     void select_am ();
     void select_fm ();
     void reset_comp_clock_time(time t);
     void comp_clock_alarm_on ();
     void comp_clock_alarm_off ();
     void comp_radio_alarm_on ();
     void comp_radio_alarm_off ();
     void set_radio_alarm_time (time);
     void set_comp_clock_alarm_time (time);
    void display_comp_clock_alarm_time ();
    void display_comp_clock_time ();
    void display_comp_radio_on_off_state ();
    void display_comp_radio_volume_level ();
    void display_comp_radio_am_channel ();
    void display_comp_radio_fm_channel ();
};


#endif CLKRAD_HPP
```

B. X_Y_POSIT CLASS METHODS

```
x_y_posit x_y_posit::operator=(x_y_posit xy)
{
  x = xy.x;
  y = xy.y;
  return *this;
}


void x_y_posit::display_position ()
{
   cout << "The XY position is X:" << x << " Y:" << y
        << ".\n";
}
```

C. TIME CLASS METHODS

```
void time::change_time (time t)
{
  if ((hours >= 0) && (hours <= 24))
    hours    = t.hours;
  else
    hours = 0;

  if ((minutes >= 0) && (minutes <= 60))
    minutes  = t.minutes;
  else
    minutes = 0;

  if ((seconds >= 0) && (seconds <= 60))
    seconds  = t.seconds;
  else
    seconds = 0;
}
```

```
time time::operator=(time t)
{
  change_time(t);
  return *this;
}


void time::display_time ()
{
  cout << "The time is " << hours << ":" << minutes <<
  ":" << seconds << "\n";
}
```

D. RADIO CLASS METHODS

```
radio::radio(x_y_posit initial_position) :
 radio_location(initial_position)
{
  radio_powered = false;
  radio_on      = off;
  fm_station    = 88.0;
  am_station    = 55.0;
  am_fm_switch  = fm;
  volume_level  = 1.0;
}


void radio::change_radio_position (x_y_posit
  new_position)
{
  radio_location = new_position;
}


void radio::power_radio()
{
  radio_powered = true;
}


void radio::remove_radio_power()
{
  radio_powered = false;
  radio_on      = off;
}
```

```
void radio::turn_radio_on ()
{
  radio_on = on;
}


void radio::turn_radio_off ()
{
  radio_powered = false;
}


void radio::increase_volume ()
{
  volume_level *= 2.0;
}


void radio::decrease_volume ()
{
  volume_level *= 0.5;
}


void radio::select_am()
{
  am_fm_switch = am;
}


void radio::select_fm()
{
  am_fm_switch = fm;
}


void radio::select_am_station (float channel)
{
  int valid_entry = false;
  while (!valid_entry)
  {
    if ((55.0 <= channel) && (channel <= 160.0))
    {
      valid_entry = true;
      am_station = channel;
    }
    else
    {
      cout << "Invalid entry, try again between 55 and
          160 khz.\n"
     cin >> channel;
    } //endif
  }  //endwhile
}


void radio::select_fm_station (float channel)
{
  int valid_entry = false;
  while (!valid_entry)
  {
    if ((88.0 <= channel) && (channel<= 108.0))
    {
       valid_entry = true;
       fm_station = channel;
    }
    else
    {
       cout << "Invalid entry, try again between 88 and
         108 mhz\n";
       cin >> channel;
    } //endif
  }  //endwhile
}


void radio::display_radio_on_off_state ()
{
  cout << "The radio is " << radio_on << ".\n";
}


void radio::display_volume_level ()
{
  cout << "The radio volume is " << volume_level
       <<  ".\n";
}
```

```
void radio::display_am_channel ()
{
   cout << "The am station is " << am_station << ".\n";
}


void radio::display_fm_channel ()
{
  cout << "The fm station is " << fm_station << ".\n";
}


void radio::display_radio_position ()
{
   radio_location.display_position();
}
```

E. ELECTRIC_CLOCK CLASS METHODS

```
electric_clock::electric_clock (x_y_posit
initial_position) : clock_location (initial_position),
current_time(), clock_alarm_time()
{
  clock_powered   = false;
  clock_alarm     = off;
}


void electric_clock::change_clock_position (x_y_posit
   new_position)
{
   clock_location = new_position;
}


void electric_clock::power_clock ()
{
  clock_powered = true;
}


void electric_clock::remove_clock_power()
{
  clock_powered = false;
}
```

```
void electric_clock::reset_clock_time(time t)
{
  current_time = t;
}


void electric_clock::clock_alarm_on()
{
  clock_alarm = on;
}


void electric_clock::clock_alarm_off()
{
  clock_alarm = off;
}


void electric_clock::set_clock_alarm_time(time t)
{
  clock_alarm_time = t;
}


void electric_clock::display_clock_alarm_time ()
{
  clock_alarm_time.display_time();
}


void electric_clock::display_clock_time ()
{
  current_time.display_time();
}


void electric_clock::display_clock_position ()
{
  clock_location.display_position();
}
```

F. CLOCK_RADIO CLASS METHODS

```
clock_radio::clock_radio (x_y_posit initial_position) :
  radio(initial_position), electric_clock
   (initial_position), radio_alarm_time (),
{
  clock_radio_powered = false;
  radio_alarm         = off;
}


void clock_radio::change_clk_rad_position (x_y_posit
    new_position)
{
   change_clock_position (new_position);
   change_radio_position (new_position);
}


void clock_radio::power_clock_radio ()
{
  power_clock ();
  power_radio ();
}


void clock_radio::remove_clock_radio_power ()
{
  remove_clock_power ();
  remove_radio_power ();
}


void clock_radio::set_radio_alarm_time (time t)
{
  radio_alarm_time = t;
}


void clock_radio::radio_alarm_on ()
{
  radio_alarm = on;
}
```

```
void clock_radio::radio_alarm_off ()
{
  radio_alarm = off;
}


void clock_radio::display_radio_alarm_on_off_state ()
{
  cout << "The radio alarm is " << radio_alarm << ".\n";
}


void clock_radio::display_radio_alarm_time ()
{
  radio_alarm_time.display_time();
}


void clock_radio::display_clock_radio_position ()
{
  display_clock_position();
}
```

G. COMPOSITE_CLOCK_RADIO CLASS METHODS

```
composite_clock_radio::composite_clock_radio (x_y_posit
  initial_position) : clock_one (initial_position),
  radio_one (initial_position), comp_radio_alarm_time ()
{
  comp_clk_rad_powered = off;   //false equals off
  comp_radio_alarm = off;           //false equals off
}


void composite_clock_radio::power_comp_clock_radio ()
{
  comp_clk_rad_powered = true;
  radio_one.power_radio ();
  clock_one.power_clock ();
}

void composite_clock_radio::remove_comp_clock_radio_power ()
{
   comp_clk_rad_powered = false;
   radio_one.power_radio ();
   clock_one.power_clock (); }
```

```
void composite_clock_radio::change_comp_clk_rad_position
  (x_y_posit new_position)
{
  comp_clk_rad_position = new_position;
  radio_one.change_radio_position (new_position);
  clock_one.change_clock_position (new_position);
}


void composite_clock_radio::turn_comp_radio_on ()
{
  radio_one.turn_radio_on ();
}


void composite_clock_radio::turn_comp_radio_off ()
{
  radio_one.turn_radio_off ();
}


void composite_clock_radio::increase_comp_radio_volume ()
{
   radio_one.increase_volume ();
}


void composite_clock_radio::decrease_comp_radio_volume ()
{
  radio_one.decrease_volume ();
}


void composite_clock_radio::select_am_station (float
  channel)
{
  radio_one.select_am_station (channel);
}


void composite_clock_radio::select_fm_station (float
  channel)
{
  radio_one.select_fm_station (channel);
}
```

```
void composite_clock_radio::select_am()
{
  radio_one.select_am ();
}


void composite_clock_radio::select_fm ()
{
  radio_one.select_fm ();
}


void composite_clock_radio::reset_comp_clock_time(time t)
{
  clock_one.reset_clock_time(t);
}


void composite_clock_radio::comp_clock_alarm_on ()
{
  clock_one.clock_alarm_on ();
}


void composite_clock_radio::comp_clock_alarm_off ()
{
  clock_one.clock_alarm_off ();
}


void composite_clock_radio::comp_radio_alarm_on()
{
  comp_radio_alarm = on;
}


void composite_clock_radio::comp_radio_alarm_off()
{
  comp_radio_alarm =  off;
}


void composite_clock_radio::set_radio_alarm_time (time t)
{
  comp_radio_alarm_time = t;
}
```

```
void composite_clock_radio::set_comp_clock_alarm_time
  (time t)
{
  clock_one.set_clock_alarm_time (t);
}


void composite_clock_radio::display_comp_clock_alarm_time ()
{
  clock_one.display_clock_alarm_time ();
}


void composite_clock_radio::display_comp_clock_time ()
{
  clock_one.display_clock_time ();
}


void composite_clock_radio::display_comp_radio_on_off_state ()
{
  radio_one.display_radio_on_off_state ();
}


void composite_clock_radio::display_comp_radio_volume_level ()
{
  radio_one.display_volume_level ();
}


void composite_clock_radio::display_comp_radio_am_channel ()
{
  radio_one.display_am_channel ();
}


void composite_clock_radio::display_comp_radio_fm_channel ()
{
  radio_one.display_fm_channel ();
}
```

APPENDIX B. SMALLTALK CODE

A. SMALLTALK CLASS DEFINITIONS

```
object subclass: #XyPosit
   instanceVariableNames: ' x y '
   classVariableNames: ''
   poolDictionaries: ''

object subclass: #SimpleTime
   instanceVariableNames: ' hours  minutes  seconds '
   classVariableNames: ''
   poolDictionaries: ''

object subclass: #Radio
   instanceVariableNames: ' radioLocation radioPowered
        radioOn  fmStation  amStation  amfmSwitch
       volumeLevel '
   classVariableNames: ''
   poolDictionaries: ''

radio subclass: #ElectricClock
 instanceVariableNames: ' clockPowered clockAlarm
        currentTime  clockAlarmTime '
   classVariableNames: ''
   poolDictionaries: ''

electric_clock subclass: #ClockRadio
   instanceVariableNames: ' clockRadioPowered
        radioAlarmTime  radioAlarm '
   classVariableNames: ''
   poolDictionaries: ''

object subclass: #CompositeClockRadio
   instanceVariableNames: ' clockOne  radioOne
        compClockRadioPosition compClockRadioPowered
        radioAlarmTime  radioAlarm '
   classVariableNames: ''
   poolDictionaries: ''
```

B. XYPOSIT CLASS METHODS

```
setx: xpos sety: ypos
   x := xp.
   y := yp.

displayPosition
   |input output char|
   input := Readstream on: 'The XY position is X:% Y:&'.
   output := Writestream on: String new.
   [input atEnd]
      whileFalse: [(char := input next) = $%
              ifTrue: [x printOn: output]
              ifFalse: [(char = $&)
                   ifTrue: [y printOn: output]
                   ifFalse: [output nextPut: char]]].
   ^output contents
```

C. SIMPLETIME CLASS METHODS

```
changeTimeHours: hrs minutes: mins seconds: secs
   hours   := hrs.
   minutes := mins.
   seconds := secs.

displayTime
   |input output char|
   input := Readstream on: 'The time is %:&:#'.
   output := Writestream on: String new.
   [input atEnd]
      whileFalse: [(char := input next) = $%
        ifTrue: [hours printOn: output]
        ifFalse: [(char = $&)
          ifTrue: [minutes printOn: output]
          ifFalse: [(char = $#)
            ifTrue: [seconds printOn: output]
            ifFalse: [output nextPut: char]]]]
   ^output contents
```

D. RADIO CLASS METHODS

```
initialize: initialPosition
   radioLocation  := XyPosit new.
   radioLocation := initialPosition.
   radioPowered  := 'false'.
   radioOn       := 'off'.
   fmStation     := 88.0.
   amStation     := 55.0.
   amfmSwitch    := 'am'.
   volumeLevel   :=  1.0.

changeRadioPosition: newPosition
   radioLocation := newPosition.

powerRadio
   radioPowered := 'true'.

removeRadioPower
   radioPowered := 'false'.

turnRadioOn
   (radioPowered = 'true')
      ifTrue: [radioOn  := 'on'].

turnRadioOff
   radioOn := 'off'.

increaseVolume
   volumeLevel := volumeLevel * 2.0.

decreaseVolume
   volumeLevel := volumeLevel * 0.5.

selectAm
   amfmSwitch := 'am'.

selectFm
   amfmSwitch := 'fm'.

selectAmStation: channel
   (channel >= 55.0 and: [channel <= 160.0])
        ifTrue: [amStation := channel].

selectFmStation: channel
   (channel >= 88.0 and: [channel <= 108.0])
         ifTrue: [fmStation := channel].

displayRadioOnOffState
   ^radioOn

displayVolumeLevel
   ^volumeLevel

displayAmChannel
   ^amStation

display_fm_channel
   ^fmStation
```

```
display_radio_position
   ^radioLocation displayPosition.
```

E. ELECTRICCLOCK CLASS METHODS

```
initialize: intialPosition
   super initialize: initialPosition.
   clockPowered    := 'false'.
   clockAlarm      := 'off'.
   currentTime     := time new.
   currentTime     := changeTimeHours: 0 minutes: 0
                      seconds: 0
   clockAlarmTime  := time new.
   clock_alarmTime := changeTimeHours: 0 minutes: 0
                        seconds: 0

changeClockPosition: newPosition
   RadioLocation := newPosition.

powerClock
   clockPowered := 'true'.

removeClockPower
   clockPowered := 'false'.

resetClockTime: newTime
   currentTime := newTime.

clockAlarmOn
   clockAlarm := 'on'.

clockAlarmOff
   clockAlarm := 'off'.

setClockAlarmTime: newAlarmTime
   clockAlarmTime := newAlarmTime.

displayClockAlarmTime
   ^clockAlarmTime displayTime.

displayClockTime
   ^currentTime displayTime.

display_clock_position
   ^radioLocation displayPosition.
```

F. CLOCKRADIO CLASS METHODS

```
initialize: initialPosition
   super initialize: initialPosition.
   clockRadioPowered := 'false'.
   radioAlarmTime    := SimpleTime new.
   radioAlarmTime changeTimeHours: 0 minutes: 0
                  seconds: 0
   radioAlarm := 'off'.

changeClkRadPosition: newPosition
   self changeRadioPosition: newPosition.

powerClockRadio
   clockRadioPowered := 'true'.
   self powerRadio.
   self powerClock.

removeClockRadioPower
   clockRadioPowered := 'false'.
   self removeRadioPower.
   self removeClockPower.
   self turnRadioOff.

setRadioAlarmTime: newTime
   radioAlarmTime := newTime.

radioAlarmOn
   radioAlarm := 'on'.

radioAlarmOff
   radioAlarm := 'off'.

displayRadioAlarmOnOffState
   ^radioAlarm

displayRadioAlarmTime
   ^radioAlarmTime displayTime.

displayClockRadioPosition
   ^self displayRadioPosition.
```

G. COMPOSITECLOCKRADIO CLASS METHODS

```
initialize: initialPosition
   radioOne := Radio new.
   radioOne initialize: initialPosition.
   clockOne := ElectricClock new.
   clockOne initialize: initialPosition.
   compClockRadioPosition := XyPosit new.
   compClockRadioPosition := initialPosition.
   compClockRadioPowered  := 'false'.
   radioAlarmTime     := SimpleTime new.
   radioAlarmTime changeTimeHours: 0 minutes: 0
           seconds: 0
   radioAlarm := 'off'.

powerCompClockRadio
   clockOne powerClock.
   radioOne powerRadio.
   compClockRadioPowered := 'true'.

removeCompClockRadioPower
   clockOne removeClockPower.
   radioOne removeRadioPower.
   compClockRadioPowered := 'false'.

changeCompClkRadPosition: newPosition
   clockOne changeClockPosition: newPosition.
   radioOne changeRadioPosition: newPosition.
   compClockRadioPosition := newPosition.

turnCompRadioOn
   radioOne turnRadioOn.

turnCompRadioOff
   radioOne turnRadioOff.

increaseCompRadioVolume
   radioOne increaseVolume.

decreaseCompRadioVolume
   radioOne decreaseVolume.

selectAmStation: channel
   radioOne selectAmStation: channel.

select_fm_station: channel
   radioOne selectFmStation: channel.

selectAm
   radioOne selectAm.

selectFm
   radioOne selectfm.

resetCompClockTime: newTime
   clockOne resetClockTime: newTime.
```

```
compClockAlarmOn
   clockOne clockAlarmOn.

compClockAlarmOff
   clockOne clockAlarmOff.

compRadioAlarmOn
   radioAlarm := 'on'.

compRadioAlarmOff
   radioAlarm := 'off'.

setRadioAlarmTime: newTime
   radioAlarmTime := newTime.

setCompClockAlarmTime: newTime
   clockOne setClockAlarmTime: newTime.

displayCompClockAlarmTime
   ^clockOne displayClockAlarmTime.

displayCompClockTime
   ^clockOne displayClockTime.

displayCompRadioOnOffState
   ^radioOne displayRadioOnOffState.

displayCompRadioVolumeLevel
   ^radioOne displayVolumeLevel.

displayCompRadioAmChannel
   ^radioOne displayAmChannel.

displayCompRadioFmChannel
   ^radioOne displayFmChannel.

displayCompClockRadioPosition
   ^compClockRadioPosition displayPosition.
```

APPENDIX C. CLOS CODE

A. CLOS CLASS DEFINITIONS

```
(defclass x_y_posit ()
   ((x :initform 0 :type integer)
    (y :initform 0 :type integer)))

(defclass time ()
   ((hours :initform 0 :type integer)
    (minutes :initform 0 :type integer)
    (seconds :initform 0 :type integer)))

(defclass radio ()
   ((radio_location :initform (make-instance
       'x_y_posit))
    (radio_powered :initform "false")
    (radio_on :initform "off" :type string)
    (fm_station :initform 88 :type float)
    (am_station :initform 55 :type float)
    (am_fm_switch :initform "off" :type string)
    (volume_level  :initform 1.0 :type float)))

(defclass electric_clock ()
    ((clock_location :initform (make-instance
       'x_y_posit))
     (clock_powered :initform "false" :type string)
     (clock_alarm :initform "off" :type string)
     (current_time :initform (make-instance 'time))
     (clock_alarm_time :initform (make-instance 'time)))

(defclass clock_radio (radio electric_clock)
   ((clock_radio_powered :initform "false" :type string)
    (radio_alarm_time :initform (make-instance 'time))
     (radio_alarm :initform "off" :type string)))

(defclass composite_clock_radio ()
   ((clock_one :initform (make-instance
        'electric_clock))
    (radio_one :initform (make-instance 'radio))
    (comp_clk_rad_positon :initform (make-instance
         x_y_posit))
    (comp_clk_rad_powered :initform "false" :type
        string )
    (comp_radio_alarm :initform "off" :type string)
    (comp_radio_alarm_time :initform (make-instance
        'time)))
```

B. X_Y_POSIT CLASS METHODS

```
(defmethod change_xyvals ((p x_y_posit) xpos ypos)
   (setf (slot-value p 'x) xpos)
   (setf (slot-value p 'y) ypos))

(defmethod display_position ((p x_y_posit))
   (format t "Current XY position is X:~a Y:~a."
    (slot-value p 'x)
    (slot-value p 'y)))
```

C. TIME CLASS METHODS

```
(defmethod change_time ((tt time) hrs mins secs)
   (setf (slot-value tt 'hours) hrs)
   (setf (slot-value tt 'minutes) mins)
   (setf (slot-value tt 'seconds) secs))

(defmethod display_time ((tt time))
   (format t "The time is ~a:~a:~a."
    (slot-value tt 'hours)
    (slot-value tt 'minutes)
    (slot-value tt 'seconds)))
```

D. RADIO CLASS METHODS

```
(defun make-object-radio ()
   (make-instance 'radio))

(defmethod change_radio_position ((r radio) xpos ypos)
   (with-slots (radio_location) r
    (change_xyvals radio_location xpos ypos)

(defmethod power_radio ((r radio))
   (with-slots (radio_powered) r
    (setf radio_powered "true")))

(defmethod remove_radio_power ((r radio))
   (with-slots (radio_powered) r
    (setf radio_powered "false")))

(defmethod turn_radio_on ((r radio))
   (with-slots (radio_on) r
    (setf radio_on "on")))

(defmethod turn_radio_off ((r radio))
   (with-slots (radio_on) r
    (setf radio_on "off")))

(defmethod increase_volume ((r radio))
   (with-slots (volume_level) r
    (setf volume_level (* volume_level 2.0))))

(defmethod decrease_volume ((r radio))
   (with-slots (volume_level) r
    (setf volume_level (* volume_level .5))))

(defmethod select_am ((r radio))
   (with-slots (am_fm_switch) r
    (setf am_fm_switch "am")))

(defmethod select_fm ((r radio))
   (with-slots (am_fm_switch) r
    (setf am_fm_switch "fm")))

(defmethod select_am_station ((r radio) channel)
   (with-slots (am_station) r
    (setf am_station channel)))

(defmethod select_fm_station ((r radio) channel)
   (with-slots (fm_station) r
    (setf fm_station channel)))

(defmethod display_radio_on_off_state ((r radio))
   (format t "The radio is ~A."
     (slot-value r 'radio_on)))

(defmethod display_volume_level ((r radio))
   (format t "The radio volume level is ~A."
     (slot-value r 'volume_level)))

(defmethod display_am_channel ((r radio))
   (format t "The am station is ~A."
    (slot-value r 'am_atation)))

(defmethod display_fm_channel ((r radio))
   (format t "The fm station is ~A."
    (slot-value r 'fm_atation)))

(defmethod display_radio_position ((r radio))
   (with-slots (radio_location) r
    (display_position radio_location)))
```

E. ELECTRIC_CLOCK CLASS METHODS

```
(defun make-object-electric_clock ()
   (make-instance 'electric_clock))

(defmethod change_clock_position ((ec electric_clock)
    xpos ypos)
   (with-slots (clock_location) ec
    (change_xyvals clock_location xpos ypos)))

(defmethod power_clock ((ec electric_clock))
   (with-slots (clock_powered) ec
    (setf clock_powered "true")))

(defmethod remove_clock_power ((ec electric_clock))
   (with-slots (clock_powered) ec
    (setf clock_powered "false")))

(defmethod reset_clock_time ((ec electric_clock) hrs
    mins secs)
   (with-slots (current_time) ec
    (change_time current_time hrs mins secs)))

(defmethod clock_alarm_on ((ec electric_clock))
   (with-slots (clock_alarm) ec
    (setf clock_alarm "on")))

(defmethod clock_alarm_off ((ec electric_clock))
   (with-slots (clock_alarm) ec
    (setf clock_alarm "off")))

(defmethod set_clock_alarm_time ((ec electric_clock) hrs
     mins secs)
   (with-slots (clock_alarm_time) ec
    (change_time clock_alarm_time hrs mins secs)))

(defmethod display_clock_alarm_time ((ec
    electric_clock))
   (with-slots (clock_alarm_time) ec
    (display_time clock_alarm_time)))

(defmethod display_clock_time ((ec electric_clock))
   (with-slots (current_time) ec
    (display_time cur_time)))

(defmethod display_clock_location ((ec electric_clock))
   (with-slots (clock_location) ec
    (display_position clock_location)))
```

F. CLOCK_RADIO CLASS METHODS

```
(defun make-object-clock_radio ()
   (make-instance 'clock_radio))

(defmethod change_clk_rad_position ((cr clock_radio)
      xpos ypos)
   (with-slots (radio_location clock_location) cr
    (change_xyvals radio_location xpos ypos)
    (change_xyvals clock_location xpos ypos)))

(defmethod power_clock_radio ((cr clock_radio))
   ((setf radio_powered "true")
    (setf clock_powered "true")))

(defmethod remove_clock_radio_power ((cr clock_radio))
   ((setf radio_powered "false")
    (setf clock_powered "false")))

(defmethod set_radio_alarm_time ((cr clock_radio) hrs
      mins secs)
   (with-slots (radio_alarm_time) cr
    (change_time radio_alarm_time hrs mins secs)))

(defmethod radio_alarm_on ((cr clock_radio))
   (with-slots (radio_alarm) cr
    (setf radio_alarm "on")))

(defmethod radio_alarm_off ((cr clock_radio))
   (with-slots (radio_alarm) cr
    (setf radio_alarm "off")))

(defmethod display_radio_alarm_on_off_state ((cr
   clock_radio))
   (format t "The radio alarm is ~A."
    (slot-value cr 'radio_alarm)))

(defmethod display_radio_alarm_time ((cr clock_radio))
   (with-slots (radio_alarm_time) cr
    (display_time radio_alarm_time)))

(defmethod display_clock_radio_position ((cr
 clock_radio))
   (with-slots (clock_location) cr
    (display_position clock_location)))
```

G. COMPOSITE_CLOCK_RADIO CLASS METHODS

```
(defun make-object-composite_clock_radio ()
   (make-instance 'composite_clock_radio))

(defmethod power_comp_clock_radio ((ccr
     composite_clock_radio))
  (with-slots (comp_clk_rad_powered clock_one radio_one)
      ccr
   (setf comp_clk_rad_powered "true")
   (power_radio radio_one)
   (power_clock clock_one)))

(defmethod remove_comp_clock_radio_power ((ccr
     composite_clock_radio))
  (with-slots (comp_clk_rad_powered clock_one radio_one)
     ccr
   (setf comp_clk_rad_powered "false")
   (remove_radio_power radio_one)
   (remove_clock_power clock_one)))

(defmethod change_comp_clk_rad_position ((ccr
    composite_clock_radio) xpos ypos)
   (with-slots (radio_one clock_one
      comp_clk_rad_position) ccr
    (change_position comp_clk_rad_position xpos ypos)
    (change_radio_position xpos ypos)))

(defmethod turn_comp_radio_on ((ccr radio))
      composite_clock_radio
   (with-slots (radio_one) ccr
    (turn_radio_on radio_one)))

(defmethod turn_comp_radio_off ((ccr
     composite_clock_radio))
   (with-slots (radio_one) ccr
    (turn_radio_off radio_one)))

(defmethod increase_comp_radio_volume ((ccr
     composite_clock_radio))
   (with-slots (radio_one) ccr
    (increase_volume radio_one)))

(defmethod decrease_comp_radio_volume ((ccr
     composite_clock_radio))
   (with-slots (radio_one) ccr
    (decrease_volume radio_one)))

(defmethod select_am_station ((ccr
     composite_clock_radio) channel)
   (with-slots (radio_one) ccr
    (slect_am_station radio_one channel)))

(defmethod select_fm_station ((ccr
     composite_clock_radio) channel)
   (with-slots (radio_one) ccr
    (select_fm_station radio_one channel)))

(defmethod select_am ((ccr composite_clock_radio))
   (with-slots (radio_one) ccr
    (select_am radio_one)))

(defmethod select_fm ((ccr composite_clock_radio))
   (with-slots (radio_one) ccr
    (select_fm radio_one)))

(defmethod reset_comp_clock_time ((ccr
     composite_clock_radio) hrs mins secs)
   (with-slots (clock_one) ccr
    (reset_clock_time clock_one hrs mins secs)))

(defmethod comp_clock_alarm_on ((ccr
     composite_clock_radio))
   (with-slots (clock_one) ccr
    (clock_alarm_on clock_one)))

(defmethod comp_clock_alarm_off ((ccr
     composite_clock_radio))
   (with-slots (clock_one) ccr
    (clock_alarm_off clock_one)))

(defmethod comp_radio_alarm_on ((ccr
   composite_clock_radio))
   (with-slots (comp_radio_alarm) ccr
    (setf comp_radio_alarm "on")))

(defmethod comp_radio_alarm_off ((ccr
     composite_clock_radio))
   (with-slots (comp_radio_alarm) ccr
    (setf comp_radio_alarm "off")))

(defmethod set_radio_alarm_time ((ccr
      composite_clock_radio) hrs mins secs)
   (with-slots (comp_radio_alarm_time) ccr
    (change_time comp_radio_alarm_time hrs mins secs)))
```

```
(defmethod set_comp_clock_alarm_time ((ccr
     composite_clock_radio) hrs mins secs)
   (with-slots (clock_one) ccr
    (set_clock_alarm_time clock_one hrs mins secs)))

(defmethod display_comp_clock_alarm_time ((ccr
     composite_clock_radio))
   (with-slots (clock_one) ccr
    (display_clock_alarm_time clock_one)))

(defmethod display_comp_clock_alarm_time ((ccr
      composite_clock_radio))
   (with-slots (clock_one) ccr
    (display_clock_alarm_time clock_one)))

(defmethod display_comp_clock_time ((ccr
      composite_clock_radio))
   (with-slots (clock_one) ccr
    (display_clock_time clock_one)))

(defmethod display_comp_radio_on_off_state ((ccr
  composite_clock_radio))
   (with-slots (radio_one) ccr
    (display_radio_on_off_state radio_one)))

(defmethod display_comp_radio_volume_level ((ccr
  composite_clock_radio))
   (with-slots (radio_one) ccr
    (display_volume_level radio_one)))

(defmethod display_comp_radio_am_channel ((ccr
    composite_clock_radio))
   (with-slots (radio_one) ccr
    (display_am_channel radio_one)))

(defmethod display_comp_radio_fm_channel ((ccr
  composite_clock_radio))
   (display_fm_channel radio_one)))
```

LIST OF REFERENCES

Arnold, P., and others, "An Evaluation of Five Object-Oriented Development Methods", Journal of Object-Oriented Programming, Focus on Analysis and Design, SIGS Publication, Inc., New York, New York, 1991, pp. 101-122.

Atkinson, L. and Atkinson, M., Using Borland C++, Que Corporation, Carmel, Indiana, 1991.

Berzins, V. and Luqi, Software Engineering with Abstractions, Addison-Wesley Publishing Company, Reading, Massachusetts, 1991.

Booch, G., Software Engineering with Ada, The Benjamin/Cummings Publishing Company Inc., Menlo Park, California, 1987.

Booch, G., Object-Oriented Design With Applications, The Benjamin/Cummings Publishing Company Inc., Menlo Park, California, 1991.

Borland International, Turbo C++ Compiler, Borland International, Inc., Scotts Valley, California, 1990.

Budd, T., An Introduction to Object-Oriented Programming, Addison-Wesley Publishing Company, Reading, Mass, 1991.

Coad, P. and Yourdan, E., Object-Oriented Design, Yourdon Press, Englewood Cliffs, New Jersey, 1991.

de Champeaux, D. and others, "Panel: Structured Analysis and Object Oriented Analysis", OOPSLA ECOOP '90 Conference Proceedings, Addison-Wesley Publishing Company, Reading, Mass, 1990, pp. 135-139.

de Paula, E. and Nelson, M., "Designing a Class Hierarchy", Technology of Object-Oriented Languages & Systems International Conference, Tools USA, Santa Barbra, California, July 29 - August 1, 1991, pp. 203-218..

Digitalk, Smalltalk/V286 Object-Oriented Programming System Tutorial and Programming Handbook, Digitalk, Inc., Los Angeles, California, 1988.

Elmasri, R. and Navathe, S., Fundamentals of Database Systems, The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, 1989.

Eckel, B., Using C++, Osborne McGraw-Hill, Inc., Berkeley, California, 1989.

Halbert, D, and O'Brien, P., "Using Types and Inheritance in Object-Oriented Programming", IEEE Software, September 1987, pp. 71-79.

Henderson-Sellers, B., and Constantine, L., "Object-Oriented Development and Functional Decomposition", Journal of Object-Oriented Programming, v. 3, No. 5, January 1991, pp. 11-16.

Ingalls, D., "A Simple Technique for Handling Multiple Polymorphism", OOPSLA Conference Proceedings, New Orleans, Louisiana, October 1 - 6, 1986, pp. 347-349.

Jacobson, I., "Industrial Development of Software with an Object-Oriented Technique", Journal of Object-Oriented Programming, v. 4, No. 1, pp. 30-40, March/April 1991.

Johnson, R. and Foote, B., "Designing Reusable Classes", Journal of Object-Oriented Programming, Focus on Analysis and Design, SIGS Publication, Inc., New York, New York, 1991, pp. 122-132.

Keene, S., Object-Oriented Programming in Common LISP, Addison-Wesley Publishing Company, Reading, Mass, 1989.

Khoshafian, S. and Copeland, G., "Object Identity", OOPSLA Conference Proceedings, New Orleans, Louisiana, October 1 - 6, 1986, pp. 406-415.

Knudsen, J., "Name Collision in Multiple Classification Hierarchies", ECOOP '88, European Conference on Object-Oriented Programming Proceedings, Oslo, Norway, August 1988, pp. 93-109.

Korson, T. and McGregor, J., "Understanding Object-Oriented: A Unifying Paradigm", Communications of the ACM, v. 33, No. 9, September 1990, pp. 40-60.

Koschmann, T., The Common Lisp Companion, John Wiley and Sons, New York, New York, 1990.

Krakowiak, S. and others, "Design and Implementation of an Object-Oriented, Strongly Typed Language for Distributed Applications", Journal of Object-Oriented Programming, v. 3, No. 3, September/October 1990, pg. 11-22.

Kuhn, T., The Structure of Scientific Revolutions, The University of Chicago Press, Chicago, Illinois, 1962.

LaLonde, W. and Pugh, J., "Subclassing /= subtyping /= Is-a", Journal of Object-Oriented Programming, v. 3, No. 5, January 1991, pp. 57-62.

Li, X., "Integration of Structured and Object-Oriented Programming", Journal of Object-Oriented Programming, Focus on Analysis and Design, SIGS Publication, Inc., New York, New York, 1991, pp. 54-60.

Lieberman, H., "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems", OOPSLA Conference Proceedings, New Orleans, Louisiana, October 1 - 6, 1986, pp. 214-224.

Loomis, M., "Integrating Objects with Relational Technology", Object Magazine, v. 1, No. 2, July/August 1991, pp. 46-60.

Meyer, B., Object-Oriented Software Construction, Prentice Hall International, New York, NY, 1988.

Micallef, J., "Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages", Journal of Object-Oriented Programming, v. 1, No. 1, April/May 1988, pp. 12-35.

Nelson, M., An Introduction to Object-Oriented Programming, NPS52-90-024, Naval Postgraduate School, Monterey, California, April 1990.

Nelson, M., "An Object-Oriented Tower of Babel", OOPS Messenger, v. 2, No. 3, July 1991, pp. 3-11.

Nelson, M., Moshell, M. and Orooji, A., "The Case for Encapsulated Inheritance", Proceedings of the 24th Annual Hawaii International Conference on System Sciences, Kauai, Hawaii, January 8-11, 1991, Vol II, pp. 219-227.

Nelson, M., Moshell, M., and Orooji, A., "A Relational Object-Oriented Management System", Ninth Annual International Phoenix Conference on Computers and Communications, Scottsdale, Arizona, March 1990, pp. 319-323.

Nierstrasz, O., "A Survey of Object-Oriented Concepts", Object-Oriented Concepts,Databases, and Applications, Kim, W. and Lochovsky, F. editors, Addison-Wesley Publishing Company, Reading, Massachusettes, 1989.

Odell, J., "Object-Oriented Analysis and Design", Journal of Object-Oriented Programming, Focus on Analysis and Design, SIGS Publication, Inc., New York, New York, 1991, pp. 74-85.

Page-Jones, M., The Practical Guide To Structured Systems Design, Yourdon Press, Englewood Cliffs, New Jersey, 1988.

Palsberg, J. and Schwartzbach, M., "Type Substitution for Object-Oriented Programming", OOPSLA ECOOP Conference Proceedings, October 21 - 25, 1990, pp. 151-160.

Perry, D. and Kaiser, G., "Adequate Testing and Object-Oriented Programming", Journal of Object-Oriented Programming, v. 2, No. 5, pp. 13-19, January/February 1990.

Pun, W. and Winder, R., "A Design Method for Object-Oriented Programming", Journal of Object-Oriented Programming, Focus on Analysis and Design, SIGS Publication, Inc., New York, New York, 1991, pp. 61-73.

Rumbaugh, J. and others, Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

Schach, S., Software Engineering, Richard D. Irwin, Inc., and Aksen Associates, Inc., Boston, Massachusetts, 1990.

Snyder, A., "Encapsulation and Inheritance in Object-Oriented Programming Languages", OOPSLA Conference Proceedings, Portland, Oregon, September 29 - October 2, 1986.

Stefik, M. and Bobrow, D., "Object-Oriented Programming: Themes and Variations", The AI Magazine, Winter 1986, v. 6, No. 4, pp. 40-62.

Stein, L., "Delegation is Inheritance", OOPSLA Conference Proceedings, Orlando, Florida, October 4 - 8, 1987, pp. 138-146.

Stroustrup, B., The C++ Programming Language, Addison-Wesley Publishing Company, Murray Hill, New Jersey, 1987.

Wasserman, A., "Object-oriented Software Development: Issues in Reuse", Journal of Object-Oriented Programming, v. 4, No. 2, May 1991, pp. 55-57.

Wegner, P., "Dimensions of Object-Based Language Design", SIGPLAN Notices, v. 22, No. 12, December 1987, pp. 168-182.

Wegner, P. and Zdonik, S., "Inheritance as an Incremental Modification or What Like is and Isn't Like", ECOOP '88, European Conference on Object-Oriented Programming Proceedings, Oslo, Norway, August 1988, pp. 55-77.

Wirfs-Brock, R. and Wilkerson, B., "Object-Oriented Design: A Responsibility-Driven Approach", OOPSLA Conference Proceedings, New Orleans, Lousisiana, October 1 - 6, 1989, pp. 71-75.

Wirfs-Brock, R., Wilkerson, B., and Wiener, L., Designing Object-Oriented Software, Prentice Hall, Englewood Cliffs, New Jersey, 1990.

Wu, T., "Benefits of Abstract Superclass", Journal of Object-Oriented Programming, v. 3, No. 6, February 1991, pp. 57-61.

Yourdon, E. and Constantine, L., Structured Design, Yourdon Press, Inc., New York, New York, 1978.

INITIAL DISTRIBUTION LIST

Defense Technical Information Center 2
Cameron Station
Alexandria, VA 221314

Dudley Knox Library 2
Code 0142
Naval Postgraduate School
Monterey, CA 93943

Chairman, Code CS 2
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

MAJ Michael Nelson 2
Code CS/NE
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

LT Alan Fink 1
6111 Madawaska Road
Bethesda, Maryland 20816

816 655